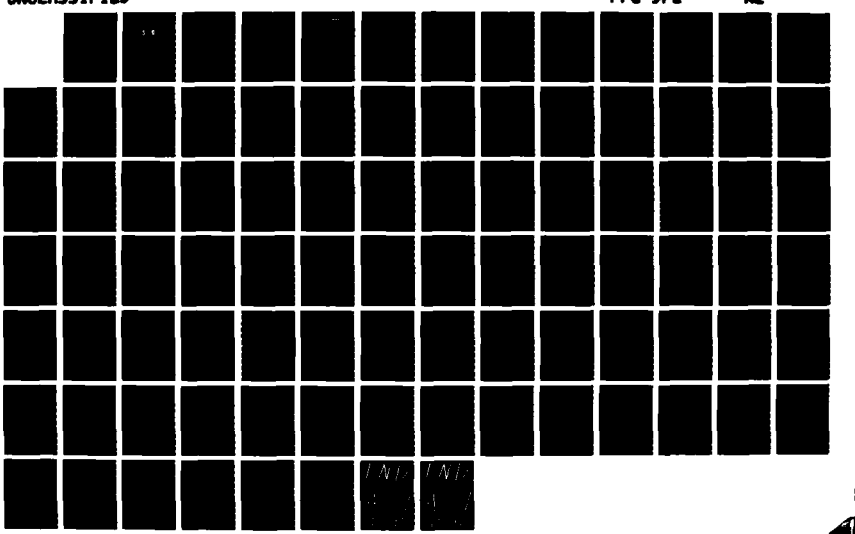
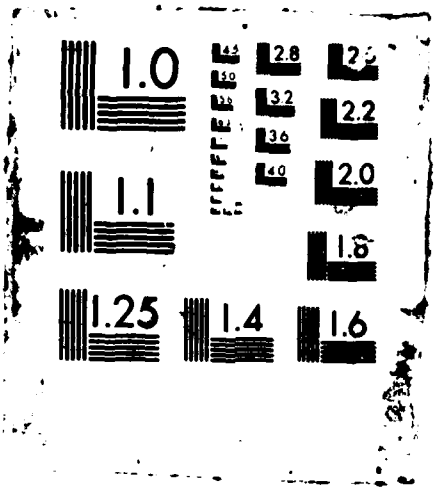
AD-A178 549 THE INFLUENCE DIAGRAMER'S TOOLBOX(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGINEERING E R DAWSON DEC 86 AFIT/GOR/MA/86D-3 1/1

UNCLASSIFIED F/G 9/2 NL




1.0
1.1
1.25
1.4
1.6
2.8
3.2
3.6
4.0
2.2
2.0
1.8

AFIT/GOR/MA

THE INFLUENCE DIAGRAMER'S TOOLBOX

THESIS

Edward R. Dawson
Captain, USAF

AFIT/GOR/MA/86D – 3

Approved for public release; distribution unlimited

AFIT/GOR/MA/86D – 3

THE INFLUENCE DIAGRAMER'S TOOLBOX

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Operations Research

Edward R. Dawson

Captain, USAF

December 1986

Approved for public release; distribution unlimited

List of Figures

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS AD-A178 579
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GOR/MA/86D-3	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION School of Engineering	6b. OFFICE SYMBOL (If applicable) AFIT/ENC
6c. ADDRESS (City, State, and ZIP Code) Air Force Institute of Technology Wright-Patterson AFB, Ohio 45433	7a. NAME OF MONITORING ORGANIZATION 7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION	8b. OFFICE SYMBOL (If applicable)
8c. ADDRESS (City, State, and ZIP Code)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER 10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
See box 19

12. PERSONAL AUTHOR(S)
Dawson, Edward Rhodes, Captain, USAF

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
MS THESIS	FROM _____ TO _____	1986 December	99

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Decision Analysis, Influence Diagrams Probabilistic Inference
12	01		

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Title: THE INFLUENCE DIAGRAMER'S TOOLBOX

Thesis Chairman: Joseph A. Tatman, Captain, USAF
Assistant Professor of Mathematics and Computer Science

Approved for public release: IAW AFR 190-1.
LYNN L. [illegible]
Dean for Research and Professional Development
Air Force Institute of Technology (ATC)
Wright-Patterson AFB OH 45433

The influence diagram is a graphical modeling language for the formulation and analysis of decision analysis and probabilistic inference problems. This research developed a foundation for a complete set of influence diagram tools as an extension to the Lisp programming language. These tools can be used in an interactive manner to solve problems modeled as influence diagrams. This research demonstrated that multidimensional arrays and frames can be used as data structures for the storage of probabilistic and nodal information.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code)	22c. OFFICE SYMBOL
Joseph A. Tatman, Captain, USAF	513-255-5533	AFIT/ENC

DD Form 1473, JUN 86 — Previous editions are obsolete. — SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

Table of Contents

Preface

The purpose of this research was to develop and implement a set of tools designed to be used for the manipulation of influence diagrams. The influence diagram is a powerful decision analysis technique which can shed light on almost any decision problem.

Given the tools and the framework developed in this thesis, the software implementor can incorporate influence diagrams into decision support systems and expert systems as well as any other appropriate application.

In performing the work and the writing of this thesis I have had a great deal of support and encouragement from others, both friends and family. I especially am thankful to my advisor, Capt Joseph A. Tatman, for being so enthusiastic and motivating in times of joy and need. I also wish to thank Maj Steve Cross for assisting and reviewing the development of my Lisp code. This thesis could not have been accomplished without the love and support of my wife, Marsha, as she strove constantly to isolate me from everyday problems. Finally, I wish to thank God for being beside me every step of the way. Truly, I have much to be thankful for as I complete this thesis.

Edward R. Dawson

Abstract

The influence diagram is a graphical modeling language for the formulation and analysis of decision analysis and probabilistic inference problems. This research developed a foundation for a complete set of influence diagram tools as an extension to the Lisp programming language. These tools can be used in an interactive manner to solve problems modeled as influence diagrams.

This research demonstrated that multidimensional arrays and frames can be used as data structures for the storage of probabilistic and nodal information, respectively. Since the array held the probabilistic data, the use of a descriptor list was introduced to lend meaning to each dimension of the multidimensional array.

The software that constitutes the toolbox, was prototyped using the APL language before being implemented in the target language, Lisp. This was advantageous in that APL was used to discover the correct array transformations that form the basis of the tools.

The results of the software development indicate that the multidimensional array is a natural and effective foundation upon which to build a complete set of influence diagram tools.

THE INFLUENCE DIAGRAMER'S TOOLBOX

I. Introduction

Background

Many decision problems have a number of interrelated uncertain variables and alternatives. Decision analysis was developed to handle problems of this type based on a firm analytical basis. One of the major techniques used in decision analysis for structuring the problem at hand is the influence diagram. An influence diagram is a graphical representation used to model a problem in terms of probabilistic variables and decisions. The diagram is a network with directed arcs and no cycles. Nodes in the diagram represent chance information and decisions. Arcs into a chance node represent probabilistic dependence upon the root node of the arc. Arcs into decision nodes represent the presence of information at the time of the decision. The diagram explictly shows information flow and probabilistic dependence. For a complete description of influence diagrams see Shacter, "Evaluating Influence Diagrams," 1986.

Historically, the idea of an influence diagram was to describe the structure of decision problems to computers for manipulation. They were first developed by researchers in the Decision Analysis Group of SRI International who were working, under contract to the Defense Advanced Research Projects Agency(DARPA), to develop computerized aids for decision analysis. To date, however, influence diagrams have been used primarily as a communication tool between the decision maker, the analyst, and functional area experts. When a solution was desired, the influence diagram was translated into the corresponding decision tree before analysis. It has been shown though that the influence diagram can be

used as the data structure for analysis and decision trees are not necessary for solution (Olmstead, 1984:12).

The influence diagram serves several purposes. It identifies and describes the interrelationships between the problem's variables. It is an important tool for communicating between the decision maker, his experts and the analyst as well as between the analyst and the computer. The influence diagram has proven to be an effective representation for both formulation and analysis of probabilistic inference and decision analysis problems.

For example, the influence diagram was used to model a toxic chemical problem. Specifically, the decision involved the determination of the carcinogenicity of the chemical and whether to ban, restrict or permit the use of the chemical. (Howard, 1984:747)

Another example is the selection of mission configurations in the Voyager Mars project conducted by NASA. The problem was to select the appropriate spacecraft configuration that would meet cost and benefit goals. Tatman showed that the simple pilot model used in the NASA analysis can be modelled as an influence diagram. (Tatman, 1985:138-9) The diagram shows clearly that the configuration of the second mission must be selected before the outcome of the first mission was known. This type of insight is a valuable commodity to the decision maker and is an inherent benefit of the influence diagram representation.

Initial research has begun to explore the use of influence diagrams as an integral part of a decision system. In his dissertation, Samuel Holtzman proposes the use of an influence diagram to embody the formal decision model as a data structure in an intelligent decision system. (Holtzman, 1985:139). He illustrated his ideas by implementing a intelligent decision system named RACHEL, a laboratory expert system designed to aid infertile couples and their doctors in the selection of medical treatment (Holtzman, 1985:155-6).

The focus of this research is the development of a sufficient set of tools for building and analyzing influence diagrams. This set of tools can then be used as building blocks in the development of software systems that use influence diagrams. Given these tools, Air Force analysts will be able to apply them to model and analyze decision problems of interest to the Air Force. Since the tools are highly portable, they are available to all Air Force analysts. Instructions for obtaining the toolbox software is given in Appendix A.

Several efforts have resulted in influence diagram solvers. SUPERID written in Lisp runs on the DEC-20. Leonard Bertrand is working on the IBM PC to build an influence diagram solver in SMALLTALK. An influence diagram solver (DAVID) has been programmed by Ross Schacter on the Apple Macintosh. All of these efforts are oriented toward a packaged environment for influence diagram analysis. This effort differs in that it provides a standard set of independent tools that may be incorporated into higher level software systems that use influence diagrams, such as decision analysis aids, intelligent decision systems, reliability analysers, and other software systems. These tools will be portable, available, and well- documented to facilitate their use as building blocks.

B. Research Problem

The goal of this thesis is the development and implementation of a highly portable set of influence diagram manipulation and analysis tools. The tools can be considered as an extension of the Lisp programming language. They will incorporate influence diagram operations into Lisp. This set of tools should facilitate the development of higher level software systems that use influence diagrams to represent probability and decision problems.

C. Research Objectives

To achieve the above goal, this thesis will develop and implement influence diagram manipulation tools as an extension of Common Lisp that are: highly portable, robust and

modifiable. Portability will be insured by the use of a development language that is implemented on a wide range of computers: from micros to mainframes. The second characteristic, robustness, is an inherent characteristic of the software itself. For a tool to be considered robust it should function in a uniform way on structures of varying degrees of complexity. The modifiability characteristic is incorporated by the use of a language that is widely accepted and by well documented code.

The various data structure alternatives that can be used to represent an influence diagram in a computer language must be explored. Also, the basic set of tools or building blocks that should be included in the toolbox must be decided upon.

It will be necessary to determine the proper software development environment. Specifically, the implementation language chosen should be capable of handling recursion, frame-based knowledge representations, and multidimensional array manipulations. Most importantly, the language chosen must allow the influence diagram tools to be broadly used in building software systems that utilize influence diagram concepts. The language, while developed on an IBM compatible, must be portable to minicomputers (such as the VAX 11/780), other microcomputers, and computers optimized for development of artificial intelligence systems. The use of the toolbox must be demonstrated by a simple application.

D. Scope

This research will be limited to influence diagrams that contain only chance nodes. Given the data and programming structures developed in this process it will not be difficult to incorporate deterministic and value nodes.

E. Summary of Results

This thesis has shown the implementation of influence diagram tools to be a viable and valuable effort. With the foundations of the software environment, simple influence diagrams can be solved interactively by the analyst.

The use of multidimensional arrays was shown to be an effective and intuitive structure for storage of the relevant probabilistic data. Further, the frame structure has been used to represent an influence diagram in a straight forward and easily understood manner.

The development of these tools is significant for three reasons. First, the tools will allow the resolution of an influence diagram down to some target diagram. For example, an arc reversal corresponds to performing probabilistic inference via an application of Bayes' rule. As the analyst uses the tools, he has full access to any intermediate results and can gain insight as the problem is being solved. Secondly, the tools can be modified to reflect some theoretically new influence procedure and then used to experiment on various problems. Lastly, the tools may be incorporated into higher level systems such as expert systems that model uncertainty.

The algorithms developed in this thesis, because of the use of multidimensional arrays, are a straight-forward translation of mathematical theory into computer code for manipulating discrete probabilistic data. Because of this fact, the code for the tools is succinct and simple.

The following chapter will introduce influence diagrams and the permitted transformations of the influence diagram before discussing the characteristics required of the software used to implement the influence diagram tools. The third chapter then presents the software implementation developed for this thesis including the data structure and the applicable functions defined in Lisp. The fourth chapter, will illustrate the influence diagram tools developed in this thesis by the solution of a reliability problem. Lastly, conclusions and further research guidance is given in the final chapter.

II. Influence Diagrams

The chapter will introduce the decision analysis probability inference environment as it relates to this research. After describing the types of problems, the mathematical manipulations allowed on influence diagrams will be covered. The final section of this chapter will detail the requirements that the software must satisfy.

A. Environment

Many decision problems are characterized by a set of uncertain events known as state variables. A state variable's uncertainty is defined by the use of probabilities to describe the likelihood of possible outcomes. It is often necessary to know the cumulative effect of all state variables to the possible outcomes. The cumulative effect can be found by forming the joint density of the desired state variables.

For example, if a, b, and c are the state variables and S represents the current state of information, then the joint density is described by {a,b,c | S}. The joint density of n state variables can be expanded as n! different products of conditional densities. For example,

$$\{a,b,c \mid S\} = \{c \mid a,b,S\}\ \{b \mid a,S\}\ \{a \mid S\}. \quad (1)$$

If it is known that

$$\{b \mid a,S\} = \{b \mid S\} \quad (2)$$

then the joint density can be written as:

$$\{a,b,c \mid S\} = \{c \mid a,b,S\}\ \{b \mid S\}\ \{a \mid S\}. \quad (3)$$

This expansion reduces the level of effort needed to solve the decision problem containing these state variables because it allows the conditional independence between the random variables to be exploited.

An influence diagram is a graphical representation of the decision problem in which it is modelled the relevant conditioning probabilities. In the influence diagram, circles represent state variables and arcs represent pairs of conditioning and conditioned variables. For example, (1) can be modelled as shown in Fig. 1.

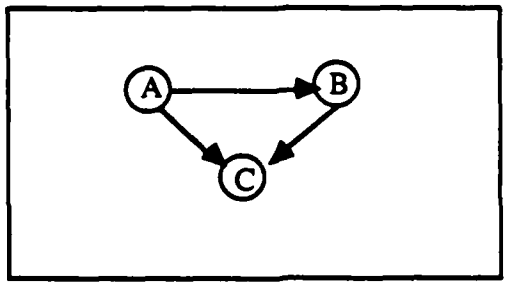


Fig. 1 Initial Diagram

This is an identity influence diagram because it corresponds in a one-to-one manner to the expansion identity and it is always a correct representation of the joint density of the state variables. Using the fact (2) gives the modified joint density (3) which is described in the following influence diagram (Fig. 2).

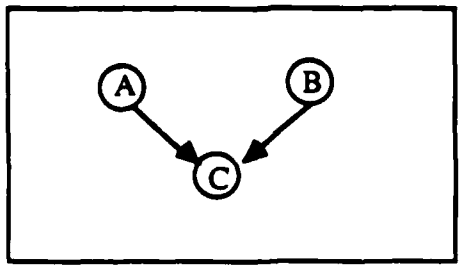


Fig. 2 Modified Diagram

The influence diagram, then, shows in a readily understandable graph the conditional dependence and, more importantly, the conditional independence among the variables.

Since each term of an expansion or identity equation represents the associated probabilistic data and each node in the influence diagram represents only one term in the expansion, the node can hold the probabilistic data associated with the term of the

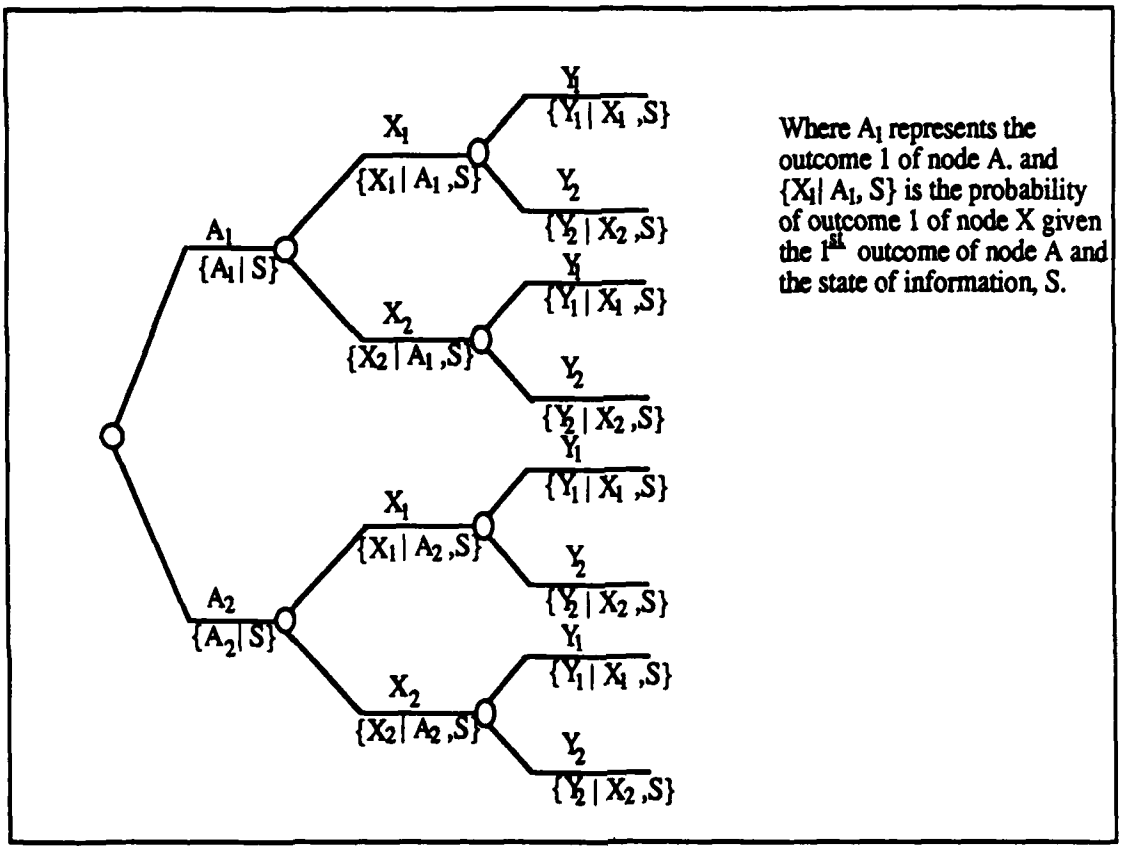


Fig. 3 Example Probability Tree

expansion. As will be shown later, this data can be stored as a multidimensional array with each dimension representing the influence from a conditioning variable and the last dimension represents the possible outcomes of the node.

The Relationship Between Influence Diagrams and Trees.

The standard technique of decision analysis is the decision tree. The structure of this tree is a representation of the decision problem and is arranged to reflect the modelling of the problem. The decision tree is composed of a decision, various alternatives, chance nodes, outcomes of the chance nodes and values. Since this thesis is concerned only with modelling the chance events of a decision problem, the preceeding Fig. 3 shows the basic elements of a probability tree.

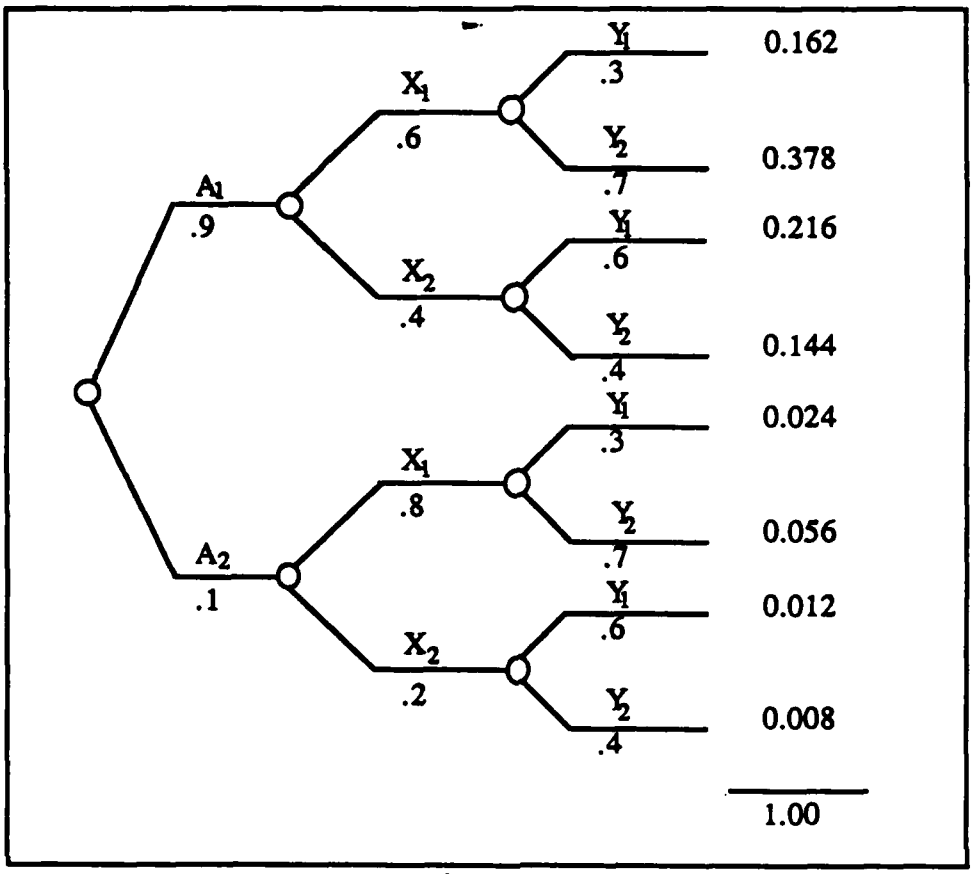


Fig. 4 Numerical Example — Tree

Notice that there are three levels of the tree corresponding to chance nodes A, X, and Y. To solve the tree, a procedure known as "rollback" is used. This procedure will give the composite effect of the probabilities along each sub-branch of the tree. This composite effect is the joint density of a sub-branch and is formed by multiplying the probabilities along the branch.

Using the structure in Fig. 3, the solution of a numerical example of the tree is shown in Fig. 4.

Notice the set of numbers on the right hand side of the tree. These numbers are the joint density of the tree and they describe the probability of each possible outcome. For instance, the top-most number is found in this way.

$$\{A_1, X_1, Y_1 \mid S\} = \{A_1 \mid S\}\{X_1 \mid A_1, S\}\{Y_1 \mid X_1, A_1, S\}$$

$$0.162 = (.9)(.6)(.3)$$

That is, the joint probability that the outcome of node A is one, the outcome of node X is one and the outcome of node Y is one equals 0.162.

There are two important points about the above example problem that need to be stated. First, notice that there are three levels of the tree. These levels correspond to the chance events A, X, and Y. Secondly, the probabilities of the outcomes of node X vary according to the probabilities of the outcomes of node A. Most importantly, also notice that while the probabilities of the outcomes of Y vary with the outcome of X, they do not vary according to the outcome of node A. Notice the numbers that are the probabilites of the outcomes of node Y in the top four sub-branches are the same in the bottom four sub-branches.

The corresponding influence diagram for the example is shown in Fig. 5.

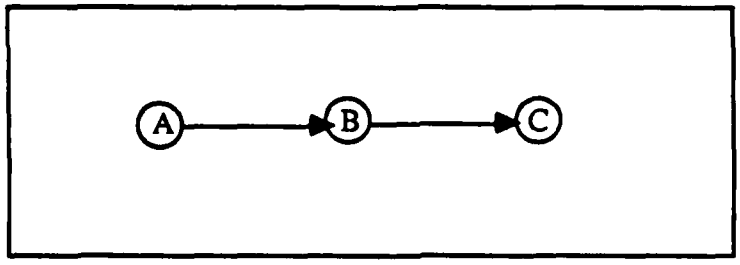


Fig. 5 Example Influence Diagram

This diagram denotes the three chance events, A, X, and Y and the "influence" on each node. The circles represent the event and the arc denotes the influence. The root node

of the arc is known as the conditioning variable and the head of the arc is the conditioned variable. Thus, the influence diagram is a compact representation of the set of variables that condition the expansion of a joint probability density. In essence the previous influence diagram describes the following equation:

$$\{A, X, Y \mid S\} = \{A \mid S\}\{X \mid A,S\}\{Y \mid X,S\}$$

The three terms on the right relate to the three nodes of the influence diagram. Since the nodes are the repository of the probabilistic information, any automated manipulation algorithm will need to relate the storage of probabilistic data to the chance nodes. For example, the data of the above example can be described by the following arrays:

Outcomes

1	2
.9	.1

X

A	1	2
1	.6	.4
2	.8	.2

Y

X	1	2
1	.3	.7
2	.6	.4

where the possible outcomes are used to index the array. For example, the probability the outcome of Y is 1 given the outcome of X to be 1 is 0.3.

In fact, $\{X \mid S\}$ can be found by performing the following array multiplication:

Prob of X given S = AX

$$[.62 \quad .38] = [.9 \quad .1] \begin{bmatrix} .6 & .4 \\ .8 & .2 \end{bmatrix}$$

So the probability of X_1 is 0.62. Notice, the left hand side is not conditioned by the outcomes of node A. In fact, the probabilistic data reflect the influence from node A. Also, since arrays are used to represent the data, the array multiplication operation can be used to find the needed information.

Comparison of the Manipulation of a Decision Tree and the Corresponding Influence Diagram

Since an influence diagram can be considered as the equivalent of the decision tree, the influence diagram can be used to solve probabilistic problems without using a decision tree.

For example, a common operation in the decision tree domain is to switch two adjacent levels of the decision tree. Since the two levels are adjacent, a probabilistic influence exists between the two levels. In the probability calculus, the above operation is an application of Bayes' rule, i.e. if Y conditions X then,

$$\{ X \mid Y, S\} = \frac{\{ X,Y \mid S\}}{\{ Y|S \}}$$

where $\{Y \mid S\}$ can be found by

$$\{Y \mid S\} = \Sigma_X \{X,Y \mid S\}.$$

In the related influence diagram, this operation is shown by reversing an influence arc between two nodes.

Using our example, the operation is demonstrated below in terms of the decision tree and the influence diagram. For the reader's convenience, the initial state is redisplayed here in (Fig. 6).

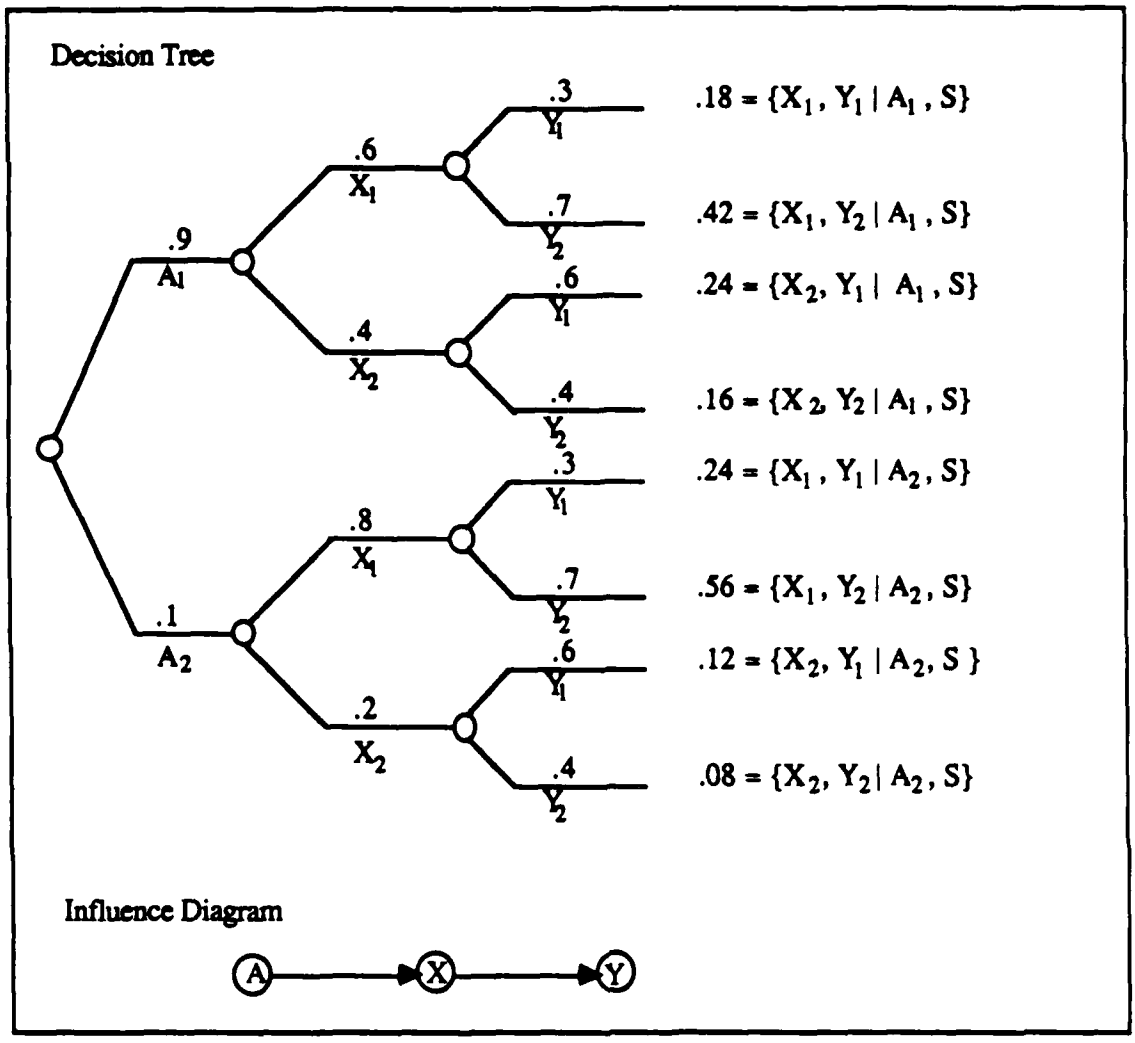


Fig. 6 Initial Tree and Diagram

Since the X and Y levels of the decision tree are to be reversed, the joint density of X and Y is shown as the right-hand side of the tree. The next step is to find $\{Y \mid A,S\}$. We want:

$$\{Y_1 \mid A_1, S\} = \{X_1, Y_1 \mid A_1, S\} + \{X_2, Y_1 \mid A_1, S\}.$$

$$= .18 + .24 = .42$$

Notice that we have summed over X. Likewise, the following terms are found:

$$\{Y_2 \mid A_1,S\} = .58$$

$$\{Y_1 \mid A_2,S\} = .36$$

$$\{Y_2 \mid A_2,S\} = .64$$

The decision tree is now updated to Fig. 7

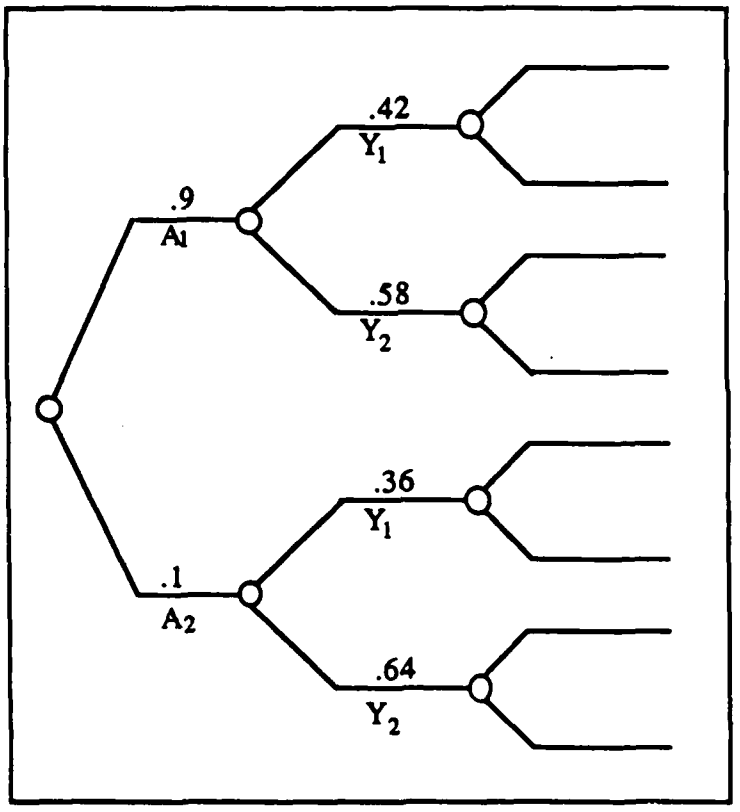


Fig. 7 Interim Tree

Since Y is now influenced by A, the interim influence diagram is Fig. 8.

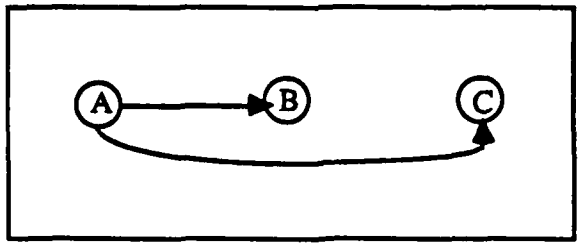


Fig. 8 Interim Influence Diagram

Now Bayes' Rule is applied to find {X | Y,A,S}.

For example, we want.

$$\{X_1, \mid Y_1 , A_1, S\} = \frac{\{ X_1, Y_1 \mid A_1, S\}}{\{Y_1 \mid A_1, S\}}$$

$$= .18/.42 \quad = 0.429$$

Continuing in this manner, the resulting decision tree is:

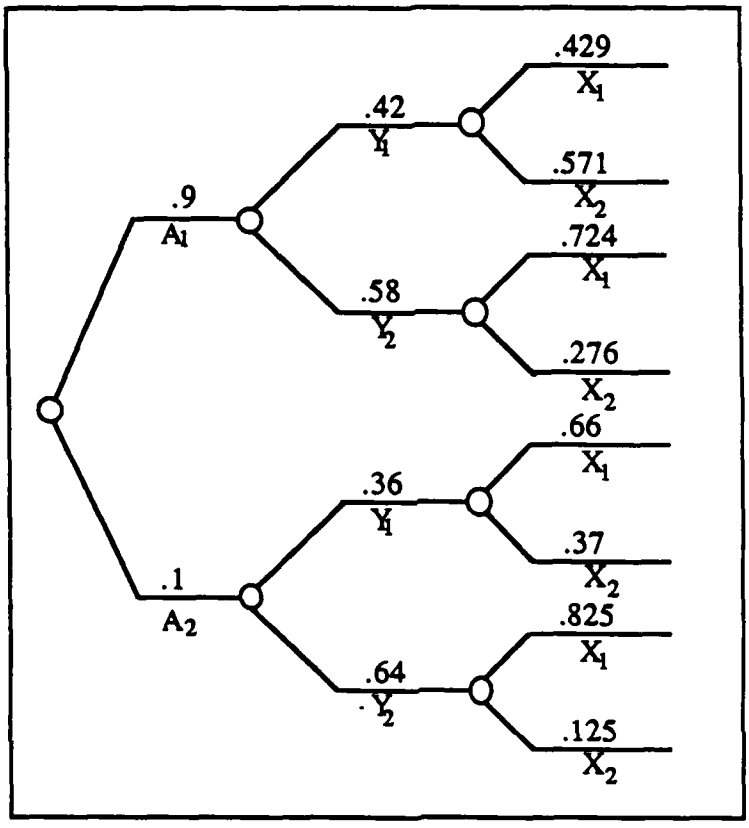


Fig. 9 Final Tree

The corresponding influence diagram is:

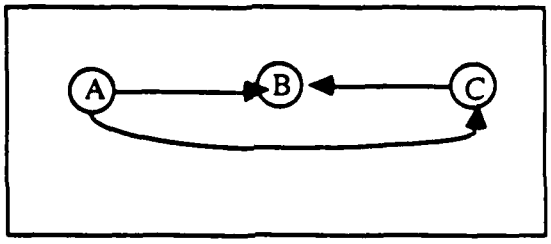


Fig. 10 Final Influence Diagram

where the node arrays for Fig. 10 are:

A	
1	2
.9	.1

X

A	Y	1	2
1	1	.429	.571
1	2	.724	.276
2	1	.66	.37
2	2	.825	.125

Y

A	1	2
1	.42	.58
2	.36	.64

The operation is now complete. The probabilistic arrays for the influence diagram are found by manipulating the initial arrays in a precise manner. This process will be presented in the section describing the arc reversal implementation in Chapter 3.

To summarize, this section has shown the close relationship between decision trees and influence diagrams. It was also demonstrated that the influence diagram can be a repositiory of probabilistic data and that data can be stored in an array. Indeed, the final array for node X in the example, is a multidimensional array of rank 3 and the list (A Y X) describes the dimensions of the array. For instance, the probability that X is one when A is two and Y is one is 0.66. Notice that this probability is the (2 1 1) element of the node X's probability array.

B. Manipulations

One of the primary objectives of this research is to show that the operations of probability theory as applied to influence diagrams can be effectively implemented using multidimensional arrays. By applying or transforming the influence diagram, insight is gained in terms of probabilistic information.

1. Terminology

Before proceeding to describe the relevant transformations of the influence diagram and its associated data, it is necessary to define the relevant terminology. An influence

diagram is a directed network characterized by a set of elements and a subset of the possible ordered pairs of the elements. Each node is an element and each arc represents an ordered pair of elements. Given a node, x, the following terms are defined:

PX = the predecessors of x are those nodes that have a directed path to x.

DPX = the direct predecessors of x are the nodes that have an arc directly to x.

SX = the successors of x are those nodes that have a directed path from x.

DSX = the direct successors of x are the nodes that have an arc directly from x.

Other combinations have similar meanings, i.e., PSX is the set of predecessors of the successors of the node x.

There are four basic operations allowed on chance nodes: reversing the arc between two nodes, removal of a node, merging two nodes, and splitting one node into two nodes. These operations will be discussed in the following sections.

2. Reversing Arcs

The reversal of an influence arc corresponds to the application of Bayes' rule in probability theory. The situation prior to the reversal is shown in the example diagram below:

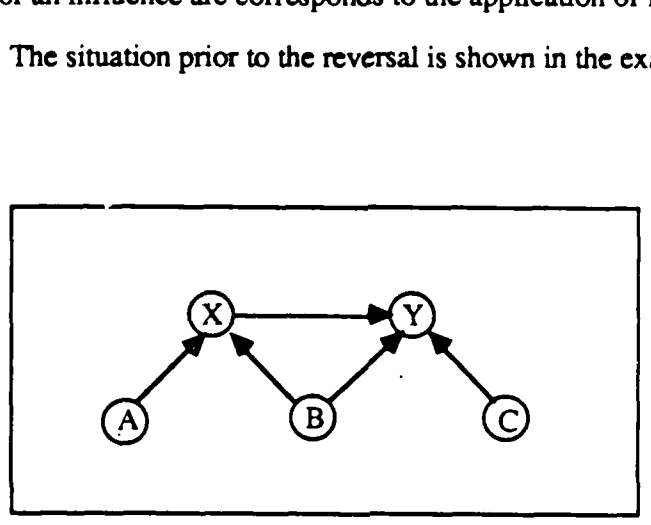


Fig. 11 Initial Influence Diagram

where PX = (a b)

PY = (x b c)

The arc to be reversed in this example is between node x and node y. Notice that no successor of x may be a predecessor of y. If this were the case, a cycle would be created by the reversal of the arc between x and y. This means the reversal can be performed only if $SX \cap PY = \emptyset$.

Corresponding to the above diagram is the following joint density:

$$\{a,b,c,x,y \mid S\} = \{a \mid S\}\{b \mid S\}\{c \mid S\}\{x \mid a,b,S\}\{y \mid x,b,c,S\} \quad (4)$$

Using the above defined terminology, (4) is rewritten as:

$$\{a,b,c,x,y \mid S\} = \{a \mid S\}\{b \mid S\}\{c \mid S\}\{x \mid PX,S\}\{y \mid PY,S\}.$$

$$= \{a \mid S\}\{b \mid S\}\{c \mid S\}\{x \mid PX,S\}\{y \mid x,PY\sim x,S\}. \quad (5)$$

where " ~ " denotes set difference.

To do the actual reversal requires three steps. First, the joint density of x and y is formed by the product of the two associated terms:

$$\{x,y \mid PXY,S\} = \{x \mid PX,S\}\{y \mid x,PY\sim x,S\}. \quad (6)$$

where PXY is the set of direct predecessors of either x or y.

The second step is accomplished to get

$$\{y \mid PXY,S\} = \Sigma_x \{x,y \mid PXY,S\} \quad (7)$$

The third step is the application of Bayes' Rule:

$$\{x \mid y,PXY,S\} = \{x,y \mid PXY,S\}/\{y \mid PXY,S\}. \quad (8)$$

Now equation (5) becomes:

$$\{a,b,c,x,y \mid S\} = \{a \mid S\}\{b \mid S\}\{c \mid S\}\{y \mid PXY,S\}\{x \mid y,PXY,S\} \quad (9)$$

where PXY = (a b c).

The influence diagram for equation (9) is in Fig. 12.

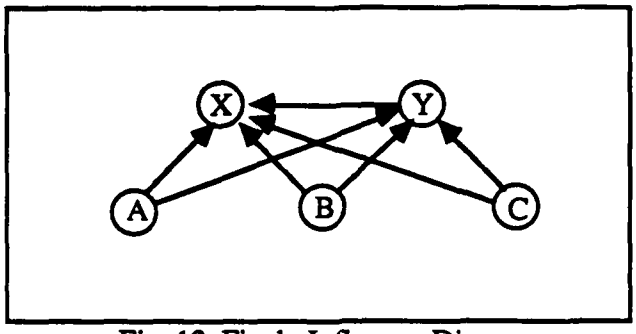


Fig. 12 Final Influence Diagram

Comparing this diagram with the previous influence diagram, the result of the arc reversal is two-fold:

1. Influences (arcs) have been added from all direct predecessors of x to y.
2. Arcs have been added from all direct predecessors of y to x.

Notice that if the arc reversal is again applied on the arc between y and x, no arcs will be added since PX~y = PY.

3. Removing a Node

The removal of a node represents summing out the relevant state variable from the joint density array of each of the node's successors. Let x represent the node to be removed. A necessary precondition of this operation is the intersection of the set of predecessors of the successors of x and the set of the successors of the successors of x must be the empty set. This rule must be met so that no cycles will be created as a result of the removal operation. After this condition has been met, the first step is to form the relevant joint density by the following equation:

$$\{x,SX \mid PX \cup PSX\sim x,S\} = \{SX \mid PSX,S\}\ \{x \mid PX,S\}.$$

Once the joint density has been formed, the next step is to sum out the influence from node x. The following equation describes this step:

$$\Sigma_x \{x,SX \mid PX \cup PSX\sim x,S\} = \Sigma_x \{x,SX \mid PX_{usx},S\} = \{SX \mid PX_{usx},S\}$$

The final step is to expand the previous expression into terms representing each of the successors of node x. If node x had only one successor, this step is not necessary.

Notice that an arc is added from every direct predecessor to every direct successor of x. Also, an arc will be added from every direct predecessor of every direct successor of x to every other direct successor of x. An influence arc must also be added between every pair of direct successor nodes of x while maintaining the previous order of conditioning variables.

The removal operation will be described below in terms of removing node g into node h of the example diagram shown below.

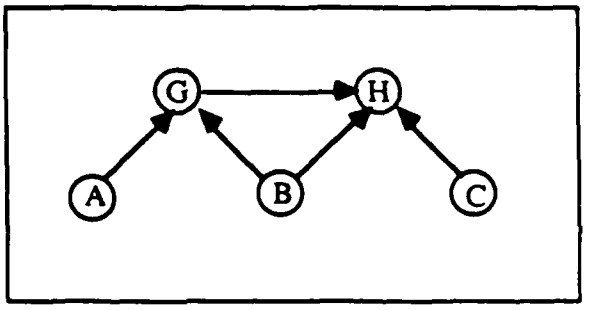


Fig. 13 Node Removal

Since there is no successors of node h, the precondition has been satisified. The three basic steps are outlined below.

1. Form the joint density:

   $\{g,h \mid a,b,c,S\} = \{h \mid b,c,S\}\ \{g \mid a,b,S\}$.

   Since node g and node h do not have the same set of conditioning variables, arcs must be added. The following expression shows that arcs have been added from node a to node h and from node c to node g:

   $\{g,h \mid a,b,c,S\} = \{h \mid a,b,c,S\}\ \{g \mid a,b,c,S\}$.

2. Form the new array for node h by summing out g from the joint density:

   $AH = \sum_g \{g,h \mid a,b,c,S\} = \{h \mid a,b,c,S\}$.

3. Since there is only one successor to node g, no expansion is necessary.

The operations of node removal and arc reversal are the only manipulations necessary to reduce an influence diagram to a single node. This is a consequence of the fact that every influence diagram has at least one conditionally independent node (i.e., no direct predecessors) and at least one node with no direct successors.

However, in many decision problems, it is desired to reduce the influence diagram to some specified subset of nodes and influences. To facilitate this goal, two more operations are needed. These operations are: merging two nodes and splitting a node into two nodes. These operations will be summarized in the next two sections. The reader is encouraged to refer to Olmstead, p18-21 for further information.

4. Merging Nodes

When two nodes are merged, say node g and node h, an influence to g or h will now be an influence to the combined node, g-h. Also an influence from either g or h will be an influence from the merge node, g-h. A precondition of this operation is that the resulting graph must have no loops, i.e., the intersection of PGH and SGH equals the empty set.

5. Splitting A Node Into Two Nodes

This operation can be considered the inverse of merging two nodes. Each influence to or from the candidate node will require an influence to or from each of the two resulting nodes.

For example, if the node g is to be split, then everywhere g appears it must be replaced by (g_1,g_2) and the term $\{g \mid PG,...\}$ must be split into the product of two terms, one for each of the resulting nodes. Also, an inference will be needed between the two resulting nodes.

It is not clear what the two terms are that must be formed from the above product. This idea will be discussed in the section on suggestions for further research.

No preconditions need to be met for the splitting operation.

C. Software Requirements

This thesis effort has been directed to implement the Influence Diagrammer's Toolbox for Air Force analyst's use in addressing decision problems. Since decision analysts have access to a variety of computer resources, it is necessary that the software developed be implementable or portable between a diverse set computer operating systems and hardware. This requirement also means that the language chosen for implementation should be a standard language that is widely avaliable. The toolbox can also be considered standard in that the influence diagram transformations are recognized as mathematically precise and sufficient for the solution of the influence diagram. These transformations are value-preserving reductions becasuse no information is lost in the transition to the resulting influence diagram.

A specific tool should also be robust. By "robust", it is meant that the tool should be able to handle diverse variations of the given nodes. For example, the arc reversal tool

should be able to handle nodes that have different number of outcomes, predecessor lists that are not equal, and arrays of different rank.

The tools developed should be modifiable since they may be applied to influence diagrams which model a specific decision problem. If the modelled probelm should require some specific structure or consideration, the tool should be amenable to modification to reflect the new characteristic of the problem. The tools may also be used to explore new types of nodes and influence diagram formulations that have been hypothesized to improve methods of solution for decison problems.

Because the software tools must be portable, robust and modifiable, certain characteristics of the development and/or the implementation language are necessary. Since the influence diagram contains all relevant data, the software environment should to able to support a frame or frame-like data structure that is easily accessible and understandable. Furthermore, the language chosen will feature multidimensional arrays as a primitive data structure for the storage of probabilistic data.

Because of the portability requirement, the language should be a language commonly available and implemented on a variety of computers. Modifiability dictates the selection of a language that is easily understood (i.e., not cryptic) and will allow a structured programming hierarchy. Both the robustness and modifiability requirements are primarily the responsibility of the programmer, but the language selection has been guided by the ease of which the requirements are met.

The language chosen for this effort was Common Lisp. Although no commercially available version of Lisp adheres fully to the Common Lisp standard (Steele), it was determined that a version that formed a sufficient subset of Common Lisp would be satisfactory. The specification of Common Lisp prescribes the use of frames and multidimensional arrays as basic to the language. The specific software used in the development of the thesis software was IQCLISP, by Integral Quality, Inc.

III. IMPLEMENTATION — PROCESS OF DISCOVERY

This chapter introduces the influence diagram software tools. To do this, the data representation is discussed as it relates to the influence diagram, the organization of the software is presented, and the algorithm for the arc reversal is related through the application to a specific example.

A. Data Representation

Since the influence diagram has been proposed to hold all of the necessary information to represent the decision problem, the data structure chosen should mirror an influence diagram as closely as possible. The data structure should allow the grouping of data elements as the characteristics of a single node. One of the most important data elements is the probability array of the node. It is desired that multidimensional arrays be used because each dimension can be designated to a specific conditioning state variable and thus gives meaning to each element of the array. By using the multidimensional array, well- defined array accessing functions and mathematical characteristics can be exploited in the software implementation. For the above reasons, the "frame" data structure was chosen. The general frame is represented as shown below:

```
(frame
   (slot
      (facet   (value))))
```

An example of a simple influence diagram in frames is:

```
(id
    (nodea
          (type   (probability))
          (predecessors ())
```

```
                (descriptors (a))
                (name a)
                (data  (0.7 0.3)) ))
(id
    (nodeb
                (type   (probability))
                (predecessors (a))
                (descriptors (a b))
                (name b)
                (data  ((0.7 0.3)
                (0.5 0.5))) ))
```

Two points need to be made at this point. First, since the descriptor list and the data arrays are closely related, (i.e., the descriptor list "describes" the dimensions of the array), it is desired that the descriptor list be in some defined order. Since, in this implementation, the nodes have alphabetical names, alphabetical order will be imposed on the first n -1 elements of the descriptor list. Since the first n -1 elements of the descriptor list constitutes the predecessor list, the predecessor will also be maintained in alphabetical order. Secondly, the last element of the descriptor list is always the node's name. So alphabetic order is enforced on only the first n -1 elements of the descriptor list.

The next section of this thesis will describe how the toolbox software is organized.

B. Hierarchy of Software

The software is organized into three areas: Construction tools, ID tools, and Supporting functions. The Construction tools are designed to allow the user to build influence diagrams and edit or delete information stored in the nodes or relationships between nodes. ID tools are those functions that are mathematically correct transformations of the influence diagram. The ID tools include all four of the operations described in Chapter 2. The Construction tools and the ID tools are considered to be "Top-Level" tools because both sets of tools are available to the user at execution time. That is, the user is free to use the Top-Level tools to analyse a decision problem. Top- Level commands are different than the Supporting functions which are introduced next.

The Supporting functions are the software needed to implement the Top-Level Commands. Although these commands are available at runtime, they support the Top-Level commands. The software was organized in this manner so that changes to the code would be localized to the appropriate function and, therefore, the code is easily modifiable. The next section will describe the Top-Level commands.

Top-Level (User) Commands

Construction Tools

The list of tools given below represents a minimal set of operations that are needed to construct valid and representative influence diagrams. Currently, only the PRINT-NODE function has been built and tested. For developmental purposes, the influence diagram representation in a frame was loaded at runtime. An example of these test frames is given in Appendix A. A brief description of the Constructor functions are given below.

BUILD-NODE - allows the user to build a node and integrate it into the current diagram.

EDIT-NODE - allows the user to change the name, predecessor list, descriptor list, type, and data associated with a given node.

DELETE-NODE - allows the user to erase a node from the current diagram. Also prompts to move or remove current arcs (dependencies).

SHOW-ID-GRAPH - presents the current influence diagram graph.

PRINT-NODE - presents the information associated with a specified node.

ID Tools

Of the influence diagram manipulations described in Chapter 2, at present the ARCREV tool is the only tool that has been built and tested. A continuation of this thesis would center upon completion of the remainder of the influence diagram manipulation tools.

ARCREV - reverses the probabilistic inference between any pair of chance nodes.

REMOVE-NODE - removes the designated node by forming the associated joint density array.

MERGE-NODES - joins two nodes into one node.

SPLIT-NODE - divides one node into two new nodes.

Supporting Functions

All supporting functions have been written so that they bear no direct relationship to influence diagrams but are an extension of the Lisp language. The supporting functions are presented in detail in Appendix A.

C. Description of Arc Reversal Code

To implement the arc reversal as computer code, the following description illustrates the algorithm with a specific example. Before proceeding, the following characteristics are defined for the node X:

PX = A list of the direct predecessors of node X.

DX = A list of the "descriptors". The first n-1 elements correspond to PX. The n^{th} element is the outcome of node X. The descriptor list is important because it gives meaning to each dimension of the probability array.

AX = The probability array associated with node X.

An example is shown by illustrating Node B:

Node B:

PB = (A)

DB = (A B)

AB =

	B	
A	0	1
0	.6	.4
1	.5	.5

Notice that the descriptor list (A B) maps in a one-to-one manner to the dimensions of the probability array. The last dimension of the array and last element of the descriptor list represent the possible outcomes of node b. Notice also that the descriptor list can always be read backwards. The list (A B C F G), then, means that the probability array of the node g represents the probability of g, given a,b,c and f. Or in mathematical notation:

{G | A,B,C,F,S}.

For example in node b, {B=1 | A=0,S} is the (0 1) element of AB, which equals 0.4.

The combination of the descriptor list and the probability array, then, forms a concise representation which uses the conditioning variable's and the requested conditioned variable outcome as indices to the probability array. The use of outcomes as indices is an intuitive way to reference the desired data.

Because the array contains all of the relevant data, it is an effective structure in that it describes all possible outcomes as conditioned by other state variables. It is effective also because well defined array manipulations can be used to implement the four influence diagram transformations. Olmstead states, "Influence diagrams whose nodes are structured as conditioning trees (as arrays do) constitute an efficient and general form for representing decision problems." (Olmstead, 1984:96) The following discussion will cover a sample arc reversal problem with specific highlights on array manipulations and operations.

For an arc reversal to be effected, there must be two nodes, say Node G and Node H.

Node G:	PG	Node H:	PH
	DG		DH
	AG		AH

The following section will describe how arcs are added to the influence diagram prior to the arc reversal.

Add-Arcs

The arc-reversal will require that DG = PH. That is, if DG = (A B G) and PH = (B C G), then DG' = PH' = (A B C G). In other words, the direct predecessors to Node H must include all of the descriptors of Node G.

Once PX, DX, and AX are updated for both Node G and Node H, the arc-reversal can take place.

ReverseX

There are five basic steps to ReverseX. Using Node G and Node H as examples, the steps are:

1. Format AG into FORMG. This action will result in the descriptors of FORMG in the same order as the descriptors of Node H.
2. Form the joint probability array of Node G and Node H by multiplying element by element the arrays FORMG and AH.
3. Define the new AH by summing down the columns of the joint probability array.
4. Format the new AH into FORMH. The descriptors of FORMH will now be the same as those for AG.
5. Define the new AG by dividing, element by element, the joint probability array by FORMH.

The following is an example described step by step using a written description, corresponding pseudo-code, current status of the related data elements, and the resulting influence diagrams.

The initial state is:

Node G: PG, DG, AG

PG = (A B)

DG = (A B G)

AG =

	.2	.8
	.7	.3
.6	.4	
.9	.1	

AG is indexed by (plane, row, column) = (A, B, C) For example, element (0 0 0) is 0.6.

Node H: PH, DH, AH

PH = (B C G)

DH = (B C G H)

AH =

	.1	.9		.3	.7
	.3	.7		.4	.6
.6	.4		.9	.1	
.5	.5		.2	.8	

AH is indexed by (hyperplane, plane, row, column)

= (B C G H)

Nodes A, B, and C have two outcomes and have no predecessors.

The initial diagram is shown in Figure 14.

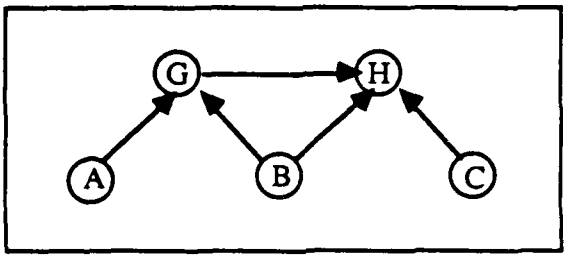


Fig. 14 Initial Influence Diagram

Arc Reversal Algorithm

1. Add arcs if necessary

Part I

a. Check to see if arcs need to be added to Node G.

Since DG does not equal PH, arcs do need to be added.

b. Pick a predecessor of H that is not a predecessor of G.

The only predecessor of H that is not a predecessor of G is node C.

c. Add the predecessor to G.

NEWARRAY = Reshape AG according to the number of outcomes associated with the incoming arc.

$$= \begin{array}{|cc|} \hline .2 & .8 \\ .7 & .3 \\ \hline \end{array} \quad \begin{array}{|cc|} \hline .2 & .8 \\ .7 & .3 \\ \hline \end{array} \quad \begin{array}{|cc|} \hline .6 & .4 \\ .9 & .1 \\ \hline \end{array} \quad \begin{array}{|cc|} \hline .6 & .4 \\ .9 & .1 \\ \hline \end{array}$$

Notice that NEWARRAY is now four-dimensional. The descriptor list that corresponds to NEWARRAY is (C A B G). It is desired that the elements of this list appear in the same order as the elements of (PG U PH). This is accomplished in the next step by transposing NEWARRAY by ANSX = (3 1 2 4). After the transposition, the descriptors of ADDX will be (A B C G).

ADDX = transpose NEWARRAY by ANSX

$$= \begin{bmatrix} .9 & .1 \\ .9 & .1 \end{bmatrix} \begin{bmatrix} .7 & .3 \\ .7 & .3 \end{bmatrix} \begin{bmatrix} .6 & .4 \\ .6 & .4 \end{bmatrix} \begin{bmatrix} .2 & .8 \\ .2 & .8 \end{bmatrix}$$

The current influence diagram showing the addition of the arc from Node C to Node G is shown in Figure 15.

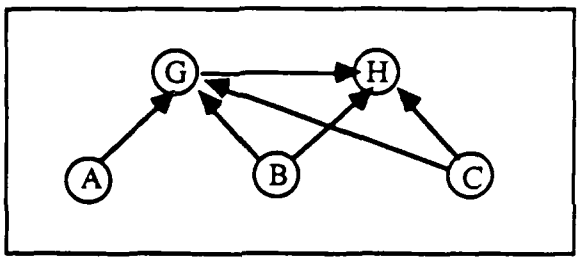


Fig. 15 Add Arc to Node G

d. Return to step 1b. and pick a new arc to be added to Node G.

No more arcs need to be added to Node G.

e. Continue until all arcs that need to be added to Node G have been added.

All necessary arcs have been added to Node G.

f. Update Node G.

AG = ADDX

PG = (LX U PG)

= (A B C)

DG = (LX U DG)

= (A B C G)

Part II

Part II will add arcs to the current successor node.

g. Check to see if arcs need to be added to Node H.

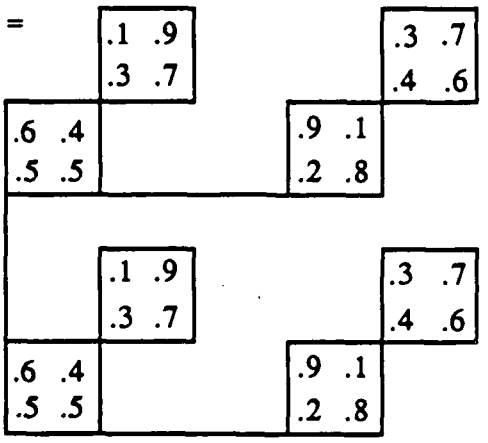
Since DG does not equal PH, arcs do need to be added.

h. Pick a predecessor of G that is not a predecessor of H.

An arc from node A needs to be added to node H.

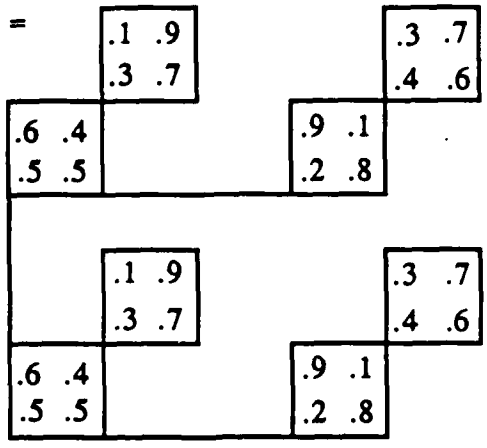
i. Add the predecessor to H.

NEWARRAY = Reshape AH according to the number of outcomes associated with the incoming arc.



Notice that NEWARRAY is now five-dimensional. The descriptor list that corresponds to NEWARRAY is (A B C G H). It is desired that the elements of this list appear in the same order as the elements of (DG U DH). This is accomplished in the next step by transposing NEWARRAY by ANSY = (1 2 3 4 5). After the transposition, the descriptors of ADDX will be (A B C G H). Even though this transposition was not necessary for this example, in general the transposition is required.

ADDY = transpose NEWARRAY by ANSY



The current influence diagram showing the addition of the arc from Node A to Node H is shown in Figure 16.

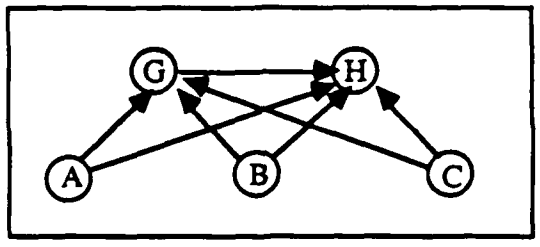


Fig. 16 Add Arc to Node H

j. Return to step 2h. and pick a new arc to be added to Node H.

No more arcs need to be added to Node H.

k. Continue until all arcs that need to be added to Node H have been added.

All necessary arcs have been added to Node H.

l. Update Node H.

AH = ADDY

PH = (LY U PH)

= (A B C G)

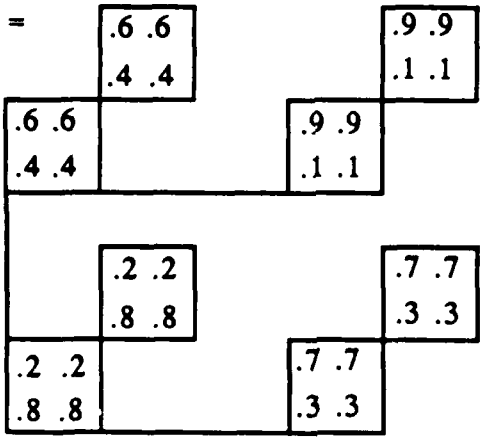
DH = (LY U DH)

= (A B C G H)

2. Reverse the Arc

a. Create FORMG by reshaping and transposing AG into the likeness of AH.

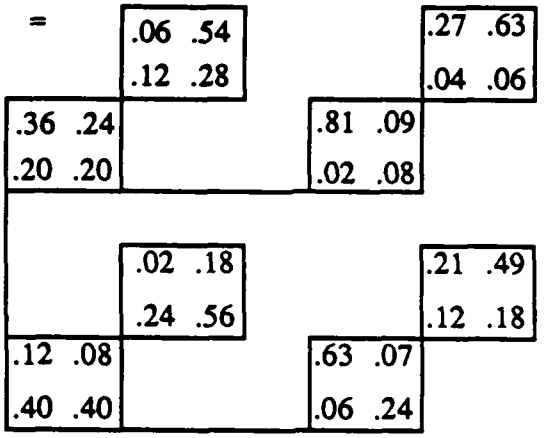
FORMG = reshape AG by the shape of AH and transpose the result so that the descriptors of FORMG will match the descriptors of AH.



Note: The above step accomplished two operations at once. First, the reshape of AG by NEWSHAPE resulted in an array that has the correct number of dimensions, but the descriptor list of the reshape operation result is (H A B C G). Before the joint density array can be computed, the elements of the above descriptor list must be in the same order as the elements of the descriptor list of Node H. The second operation will transpose the array resulting from the reshape operation into the same orientation as AH. The descriptor list of FORMG is (A B C G H).

b. Form the Joint probability array. Notice that the descriptor lists of FORMG and AH are the same.

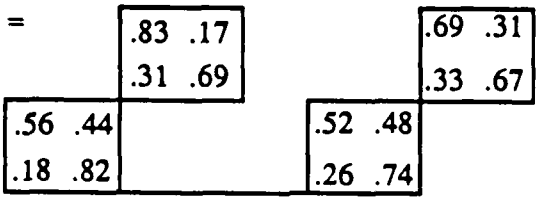
JOINT = FORMG x AH



The descriptor list of JOINT is (A B C G H).

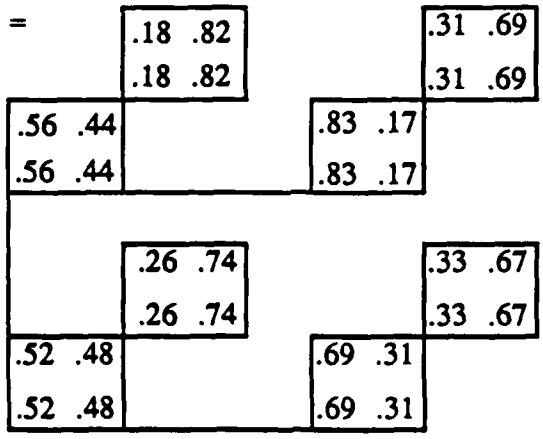
c. Create the new AH by summing down the columns of the joint probability array.

NEWAH = sum down all columns of the joint probability array. For this example, this operation will sum out the inference from Node G. The descriptor list of the array shown below is (A B C H).



d. Create FORMH by reshaping and transposing new AH into the likeness of the joint probability array.

FORMH = reshape NEWAH by NEWSHAPE and transpose the result so that the descriptors of FORMH will correspond to the descriptors of the Joint probability array.



Note: The above step accomplished two operations at once. First, the reshape of NEWAH results in an array that has the correct number of dimensions, but the descriptor list of the reshape operation result is (G A B C H). Before the new array for Node G can be computed, the elements of the above descriptor list must be in the same order as the elements of the descriptor list of JOINT. The second operation will transpose the array resulting from the reshape operation into the same orientation as the array JOINT. This is accomplished by transposing the result-array. The descriptor list of FORMH is (A B C G H).

e. Create the new AG by dividing the joint probability array by FORMH.

NEWAG = transpose (JOINT / FORMH) to place the derived outcomes of the array AG in the last dimension.

=

		.33	.67			.87	.13
		.66	.34			.91	.09
.64	.36			.98	.02		
.54	.46			.53	.47		
		.08	.92			.64	.36
		.24	.76			.73	.27
.23	.77			.91	.09		
.16	.84			.23	.77		

f. Update Node G and Node H.

AG = NEWAG

=

		.33	.67			.87	.13
		.66	.34			.91	.09
.64	.36			.98	.02		
.54	.46			.53	.47		
		.08	.92			.64	.36
		.24	.76			.73	.27
.23	.77			.91	.09		
.16	.84			.23	.77		

AH = NEWAH

$$= \begin{array}{|cc|cc|} \hline .83 & .17 & & \\ .31 & .69 & & \\ \hline & & .69 & .31 \\ & & .33 & .67 \\ \hline \end{array} \begin{array}{|cc|cc|} \hline .56 & .44 & & \\ .18 & .82 & & \\ \hline & & .52 & .48 \\ & & .26 & .74 \\ \hline \end{array}$$

PH = PG

= (A B C)

PG = (PG U (DH - PH))

= (A B C H)

DG = switch last two of DH

= (A B C H G)

DH = PG

= (A B C H)

The final influence diagram showing the reversal of the arc from Node G to Node H is shown in Figure 17.

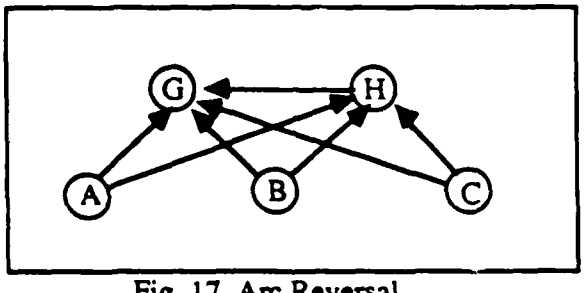


Fig. 17 Arc Reversal

Example Application of ARCREV

To show how ARCREV may be used in practice, the following example run has been provided:

After loading the Influence Diagrammer's Toolbox and performing the following command:

```
(setq current-id 'id1),
```

the following information is provided via the print-node command (and is the initial state):

```
(print-node 'nodeg)
"Node name is: "
(G)
"Node predecessors are: "
(A B)
"Node descriptors are: "
(A B G)
"Node probability array is: "
(3#A (((0.6 0.4)
       (0.9 0.1))
      ((0.2 0.8)
       (0.7 0.3)))  )

(print-node 'nodeh)
"Node name is: "
(H)
"Node predecessors are: "
(B C G)
"Node descriptors are: "
(B C G H)
"Node probability array is: "
(4#A ((((0.6  0.4)
        (0.5  0.5))
       ((0.1  0.9)
        (0.3  0.7)))
      (((0.9  0.1)
        (0.2  0.8))
       ((0.3  0.7)
        (0.4  0.6))))  )
```

So the initial state is the following influence diagram:

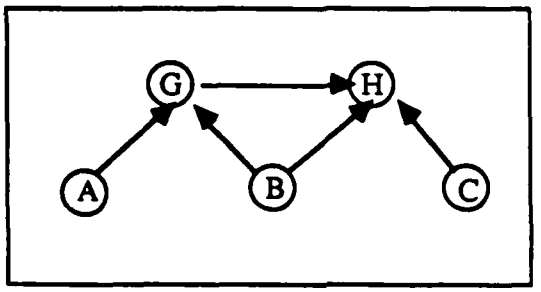


Fig. 18 Initial Influence Diagram

Now the ARCREV command is applied to the arc between node G and node H.

```
(arcrev  'nodeg  'nodeh)
(ARC REVERSAL IS COMPLETE)
```

The resulting information is presented by print-node:

```
(print-node  'nodeg)
"Node name is: "
(G)
"Node predecessors are: "
(A  B  C  H)
"Node descriptors are: "
(A  B  C  H  G)
"Node probability array is: "
   (#5A  (((((  0.64   0.36 )
               (   0.54   0.46 ))
              ((  0.33   0.67 )
               (   0.66   0.34) ))
             (((   0.98   0.02 )
                (  0.53   0.47 ))
               ((  0.87   0.13 )
                (  0.91   0.09 ))))
             ((((  0.23   0.77 )
                (  0.16   0.84 ))
               ((  0.08   0.92 )
                (  0.24   0.76 )))
              (((  0.91   0.09 )
               (   0.23   0.77 ))
              ((   0.64   0.36 )
               (   0.73   0.27 )))))    )
```

```
(print-node  'nodeh)
"Node name is: "
(H)
"Node predecessors are: "
(A  B  C)
"Node descriptors are: "
(A  B  C  H)
"Node probability array is: "
    (#4A  ((((  0.56   0.44 )
             (  0.18   0.82 ))
            ((  0.83   0.17 )
             (  0.31   0.69 )))
           (((  0.52   0.48 )
             (  0.26   0.74 ))
            ((  0.69   0.31 )
             (  0.33   0.67 ))))  )
```

So the final resulting influence diagram is:

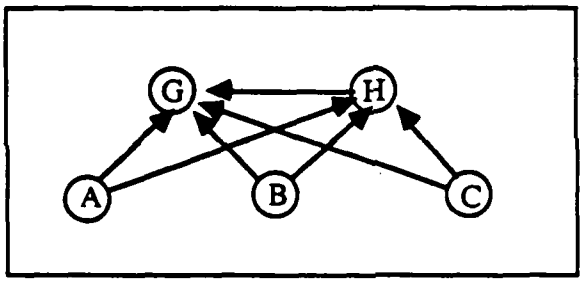


Fig. 19 Final Influence Diagram

Node Removal Algorithm

The arc reversal and node removal operations are closely related. The arc reversal is effected by forming the joint density between two adjacent nodes, summing the old predecessor out of the joint density array to get the new predecesssor, and finally getting the new successor by applying Bayes' rule. As described before, node removal entails forming the joint density array and summing out of the joint density array the variable represented by the node to be removed. Notice that node removal is the same as arc reversal through the creation of the new predecessor. In the case of node removal, this is

the updated successor. Since these two operations are so similar, the algorithms for add arcs and reversex can be used to build REMOVE-NODE.

Specifically, REMOVE-NODE could use all steps of the arc reversal algorithm through step 2b. All that remains to be done is to update the influence diagram's frame to reflect the deletion of the specified node.

The node removal tool can then be developed using the same supporting commands as ARCREV.

The remaining commands, merge-nodes and splitting a node, as shown in Chapter 2, are valid transformations of the influence diagram. The algorithms for these commands can be derived from the previously given theoretical base and should be implementable via the supporting commands in Appendix A. An obvious extension of this thesis, then, is the implementation of node removal, merge nodes, and splitting of a node operations.

IV. RELIABILITY APPLICATION

Application

This chapter will explain the application of arc reversal to a simple reliability problem.

Given the following diagram which describes a subfunction, S of a electronic circuit card. The subfunction, S, consists of two parts: a nested subfunction, E, and component D. Subfunction E, in turn is composed of components A, B, and C.

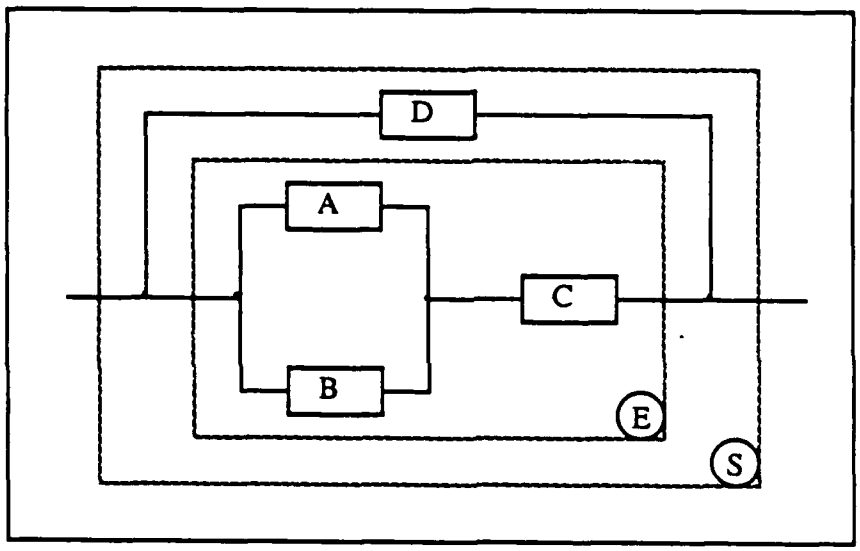


Fig. 20 Electronic Circuit Card

Each subfunction or component has two states: working or not working. The following data presents the assessed probabilities for each subfunction or component.

A	
0	1
.3	.7

B	
0	1
.6	.4

C	
0	1
.5	.5

D	
0	1
.1	.9

A	B	C	E: 0	E: 1
0	0	0	1	0
0	0	1	.9	.1
0	1	0	.8	.2
0	1	1	.4	.6
1	0	0	.7	.3
1	0	1	.4	.6
1	1	0	.4	.6
1	1	1	0	1

D	E	S: 0	S: 1
0	0	1	0
0	1	.4	.6
1	0	.5	.5
1	1	0	1

The tables may be interpreted in this way: the probability that E is not working (0) given that components A, B, C are all not working is one. The given state of a subfunction or component is used to index the probability data.

Based on the above information the following influence diagram represents the initial state:

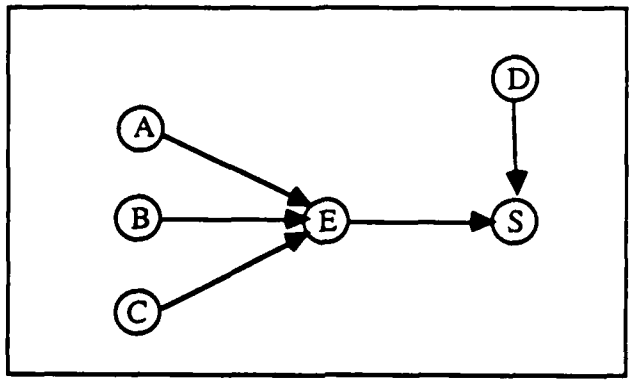


Fig. 21 Initial Influence Diagram

A possible question that could be posed is: "What is the probability that the subfunction, S, works?" To answer this question it is necessary to manipulate the influence diagram so that the node S has no predecessors. By judicious application of arc reversals, this information can be found.

Using the diagram as a guide, it can be seen that the arcs from E to S and from D to S need to be reversed. First, the arc between E and S is examined as a possible candidate for arc reversal. Notice that arcs would be added to S from A, B, and C and from D to E. Since this action will result in nodes A, B, and C becoming predecessors of S; arcs which would also need to be reversed.

Instead of reversing the arc from E to S, arc reversals will be done to remove all of the current predecessors of E. Once this is done, the only arc that will be need to be added

will be from node D to E. At that point, the arc E to S can be reversed. Notice also that the arc from D to S can be reversed without adding arcs.

So the procedure that will be followed is:

1. Reverse A to E
2. Reverse B to E
3. Reverse C to E
4. Reverse D to S
5. Reverse E to S

1. Reverse A to E. Since the predecessors of E are not all predecessors of A, arcs will be added from B and C to A. Then the arc between A and E is reversed. These actions result in the following influence diagram and updated data for A and E.

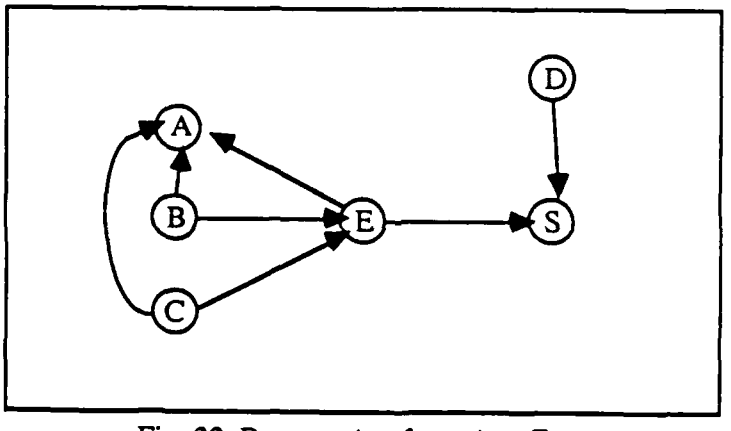


Fig. 22 Reverse Arc from A to E

A

B C E	0	1
0 0 0	.38	.62
0 0 1	.49	.51
0 1 0	.46	.54
0 1 1	1	0
1 0 0	0	1
1 0 1	.07	.93
1 1 0	.125	.875
1 1 1	.21	.79

E

B C	0	1
0 0	.79	.21
0 1	.55	.45
1 0	.52	.48
1 1	.12	.88

2. Reverse B to E. Since C is a predecessor of E but not of B, it is required that an arc be added from B to C. At this point the arc from B to E can be reversed and results in the influence diagram and updated tables below:

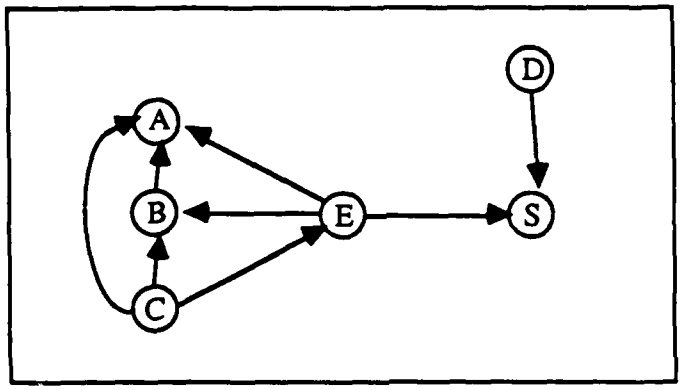


Fig. 23 Reverse Arc from B to E

B

C E	0	1
0 0	.695	.305
0 1	.873	.127
1 0	.396	.604
1 1	.434	.566

E

C	0	1
0	.682	.318
1	.378	.622

3. Reverse C to E. Since the only predecessor of E is now C itself, no arcs need to be added. Reversing the arc between C and E gives:

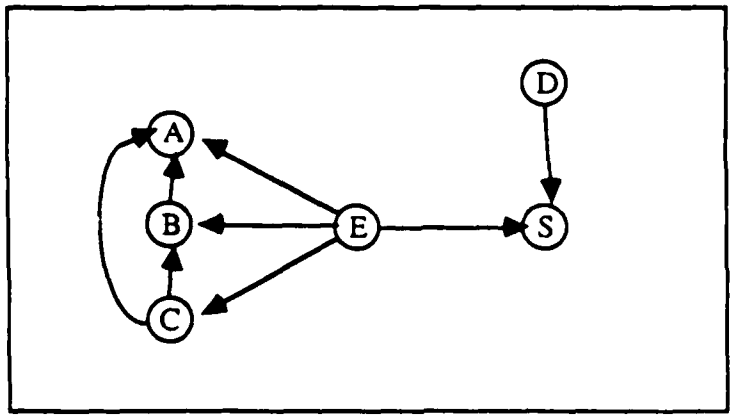


Fig. 24 Reverse Arc from C to E

The probability arrays for nodes C and E are shown below:

C

E	0	1
0	.644	.356
1	.338	.662

E

0	1
.53	.47

Notice that the probabilty of subfunction E working is 0.47. All conditioning variables have removed and E is conditionally independent at this point in the analysis.

4. Reverse D to S. Now E is a predecessor of S but not of D so it is required that an arc be added from E to D. This action facilitates the arc reversal from D to S which caused the influence diagram and data to be updated as shown below:

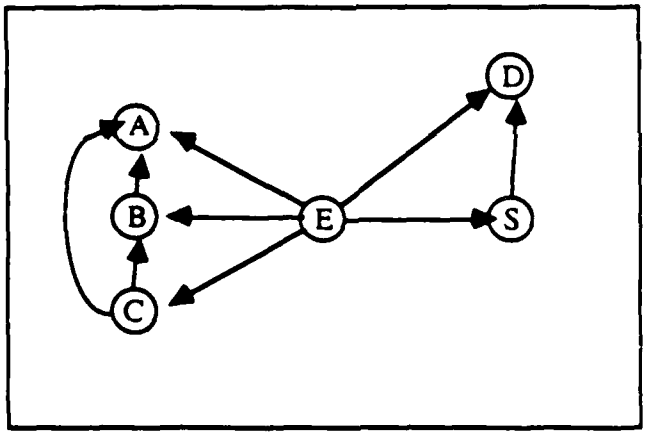


Fig. 25 Reverse Arc from D to S

D

E	S	0	1
0	0	.18	.82
0	1	0	1
1	0	1	0
1	1	.06	.94

S

E	0	1
0	.55	.45
1	.04	.96

5. Reverse E to S. No arcs need to be added since E has no predecessors and S has E as its only predecessor. This final reversal gives:

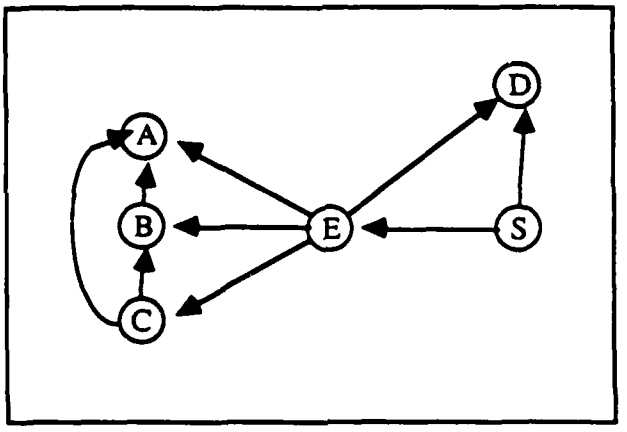


Fig. 26 Reverse Arc from E to S

and the final data is:

A

B C E	0	1
0 0 0	.38	.62
0 0 1	.49	.51
0 1 0	.46	.54
0 1 1	1	0
1 0 0	0	1
1 0 1	.07	.93
1 1 0	.125	.875
1 1 1	.21	.79

B

C E	0	1
0 0	.695	.305
0 1	.873	.127
1 0	.396	.604
1 1	.434	.566

C

E	0	1
0	.644	.356
1	.338	.662

D

E S	0	1
0 0	.18	.82
0 1	0	1
1 0	1	0
1 1	.06	.94

E

S	0	1
0	.9394	.0606
1	.3438	.6542

S

0	1
.3103	.6897

Now that the procedure is complete, it can be seen that the probability that S works is 0.6897.

Other questions of a similar nature can be answered by similar transformations of the influence diagram.

V. CONCLUSIONS AND SUGGESTIONS FOR FURTHER RESEARCH

This thesis demonstrated that the multidimensional array is a natural foundation for the development of a toolbox of influence diagram manipulation and analysis tools. This idea was used to develop and implement an algorithm to accomplish an arc reversal on two nodes of an influence diagram. Furthermore, a sample reliability problem was solved using the arc reversal tool. This chapter will present some of the issues or problems that need to be addressed and will propose some extensions to this thesis.

A. Issues

There are two major concerns that need to be addressed in any follow-on research. The first concern is the specification of the two new nodes to be created when splitting a node. How should the probabilistic array of the original node be allocated to the two new nodes? Is the probability array of each child node a duplication of the original probability array?

The second concern is involved in the removal of a node operation. If the resulting probability array has more than one conditioned state variable, the array needs to be expanded into more nodes. The question is how is the expansion to be executed and in what order are the nodes to be created. For more information on these problems see Olmstead's dissertation, "On Representing and Solving Decision Problems."

B. Extensions and Improvements

There are several extensions to the software that are necessary to have a complete environment and a robust set of tools. The extensions are:

The Constructor functions need to be completed and integrated into the environment.

It is desirable to have a one screen representation of the entire influence diagram. If the proper graphics environment can be defined, it would be optimal to represent nodes

and arcs as graphics primitives. However, portability should not be sacrificed to meet this goal.

The remaining ID tools need to be implemented: Remove Nodes, Merge Nodes, and Split Nodes.

A fast and efficient adaptation of the branch and bound routine is needed for the loop checking precondition of the arc reversal, merge nodes and node removal operations.

Add value, decision, and deterministic nodes and associated operations to the current set of software.

Once the software is completely developed, it may be used in experimental research on the optimal ordering of node removals.

The software should be ported to several computers by use of the XLISP version of Common Lisp.

As is always true in software development, the software is never complete; however, the more research done along these lines will give analysts and decision makers more capability for developing systems, based on influence diagrams, for artificial intelligence, reliability analysis and other computer-based systems.

APPENDIX A DOCUMENTED LISP CODE

This Appendix gives a listing of the source code for the Influence Diagrammer's Toolbox. A Lisp-loadable version of the code can be obtained by contacting:

Capt. Joe Tatman
AFIT/ENC
WRIGHT-PATTERSON AFB, OH 45433

To load and run the code, the target Lisp should be Common Lisp compatible and have implemented multidimentional arrays according to the Common Lisp standard.

The software contains sections describing the following in order:

1. ID tools
2. ID access functions
3. Test frames (influence diagrams).
4. Frame Manipulation Functions
5. Lisp Tools (supporting commands which are an extension to the Lisp language).

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; ID tools
;
; Arcrev will reverse the probabilistic inference between
;   two adjacent nodes. First, the nodes have arcs added,
;   if necessary. Secondly, the reversal is effected by
;   use of Bayes rule.

(defun arcrev (nodea nodeb)
 (addarcs nodea nodeb)
 (reversex nodea nodeb)
'(arc reversal is complete))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Reversex will perform Bayes' Rule on two adjacent nodes
; that have the same set of common predecessors

(defun reversex (nodea nodeb)

; This first section will load local variables with
; information from the frame for the two nodes involved.
; Arraya, pa and na represents the data array, predecessor
; list, and the name(symbol) of the node a. Likewise,
; arrayb and nb are the array and name of the node b.

  (let* ((arraya (get-prob-array nodea))
       (arrayb (get-prob-array nodeb))
       (pa (get-preds nodea))
       (na (get-name nodea))
        (nb (get-name nodeb))

; This next section will reshape array a into the likeness
; of array b. Newshapea represents the dimension list by
; which to reshape arraya. Forma is the result of
; transposing the reshape of arraya. The purpose of this
; section is to make arraya conformable to arrayb and to
; have corresponding elements aligned. Ra is simply the
; rank of the predecessor's array. Ira is a list of the
; numbers 1 through the rank of the array (ra). (list-
; rotate ra ira) provides the trans-index to transpose the
; reshaped arraya.

     (newshapea (append (last (array-dimensions arrayb))
              (array-dimensions arraya)))
     (ra (length (array-dimensions arraya)))
     (ira (iota (length (array-dimensions arrayb))))
     (forma (array-transpose (array-reshape arraya
                                 newshapea)
                (list-rotate ra ira)))
```

```
; The next line will give the joint density function
; between node a and node b.

   (joint (array-mult-elt forma arrayb))

; The next two lines will define the new array to be stored
; in the old successor node (new predecessor node). This
; is done by summing down the columns of the joint density
; array.

   (si (1- (length (array-dimensions arrayb))))
    (newab (array-sum-reduce joint 2))

; This next section will reshape newab into the likeness
; of the joint array. Newshapeb represents the dimension
; list by which to reshape newab. Formb is the result of
; transposing the reshape of newab. The purpose of this
; section is to make newab conformable to the joint density
; array and to have corresponding elements aligned. Si
; (from the previous section), y and irb are used to
; contruct the appropriate trans-index to transpose the
; reshaped newab.

   (y (list-difference (iota (length (array-dimensions
                                arrayb)))
                (list si)))
   (irb (append (list si) y))
   (newshapeb (append (last (array-dimensions arraya))
          (array-dimensions newab)))
   (formb (array-transpose (array-reshape newab
                                newshapeb)
             irb))

; This section shows the creation of the new array of the
; successor (newaa). X is the index used to realign the
; result of dividing the joint array by formb. This
; operation will ensure that the dimension describing the
; outcomes of the array will come last in the descriptor
; list.

   (x (switch-last-two (iota (length (array-dimensions
                                arrayb)))))
   (newaa (array-transpose (array-div-elt joint formb)
             x)) )
```

```
; The remainder of this function is designed to update the
; influence diagram with the new information.

   (set-descr nodea (list (switch-last-two (get-descr
                        nodeb))))
   (set-descr nodeb (list (list-union (get-preds nodea)
                              (get-name nodeb))))
   (set-prob-array nodea (list (list-union pa nb)) newaa)
           (set-prob-array nodeb (list (list pa)) newab)
     't ))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; The function addarcs will add arcs to either or both of
; two adjacent nodes. Nodea is assumed to be the
; predecessor of nodeb. When addarcs is finished, both
; nodes will have the same set of direct predecessors
; (except, of course, nodea will not have nodeb as a
; predecessor.

(defun addarcs (nodea nodeb)

; This first section will load into local variables the
; relevant information about the two nodes from the
; influence diagram.

 (let* ((pa (get-preds nodea))
       (da (get-descr nodea))
       (aa (get-prob-array nodea))
       (pb (get-preds nodeb))
       (db (get-descr nodeb))
       (ab (get-prob-array nodeb)))

; This is the stopping condition,... when the predecessor
; list of the successor is the same as the descriptor list
; of the predecessor.

    (cond ((list-equal da pb)
          '("arcs finished"))

; Lx is the list of the nodes from which arcs need to be
;   added to predecessor node.
; Ly is the list of the nodes from which arcs need to be
;   added to successor node.

         (t (let* ((lx (list-difference (list-union
                                  pa pb) da))
                  (ly (list-difference (list-union
                                  da db) db)))

; Once all of the arcs have been added to the
; predecessor node, stop and go add arcs to the
; successor node.

                  (cond ((not (equal lx nil))

; Ix represents the index of the union set which
; corresponds to the node from which an arc must be added.
; Iu is the list (1 2 3 ... # of elements in (the union of
; the predecessors of the predecessor node and the
; predecessors of the successor node)) - (IX). Ix and Iu
; are used to build ANSX, which is the trans-index used to
; transpose the data array so that the new information
```

```
; gained by adding the arc will align with the other data
; in the array.  NARCOUT is number of the incoming arc's
; outcomes (always equal to the last dimension of the array
; of the node from which an arc is being added.
;   NEWSHAPE is the new size of the predecessor array
; after the addition of the new arc.
;   ADDX then is the new array after addition of the arc.

                (let* ((ix (list-elt-index
                     (list-union pa pb)
                     (car lx)))
                   (iu (list-difference
                     (iota (length
                          (list-union
                           pa pb)))
                     (list ix)))
                  (ansx (append (list ix)
                          iu))
                  (narcout
                  (array-dimension ab
                  (1-(list-elt-index
                      pb (car lx)))))
                 (newshapea (append
                    (list narcout)
                    (array-dimensions
                      aa)))
                  (addx (array-transpose
                      (array-reshape
                       aa newshapea)
                      ansx)))

; The next two lines update the frame with the new
; information of the predecessor node and calls addarcs to
; add the next arc.

                (set-prob-array
                 nodea
                (list
                  (list-union
                   (list (car lx)) pa))
                 addx)
                (set-descr
                 nodea
                 (list
                  (list-union
                   (list (car lx)) da)))
                 (addarcs nodea nodeb)))
```

```
; The remainder of this function add arcs to the successor
; node in the same manner as above.

                (t (let* ((iy (list-elt-index
                        (list-union da db)
                        (car ly)))
                      (iu (list-difference
                        (iota
                          (length
                           (list-union
                             da db)))
                         (list iy)))
                      (ansy (append (list iy)
                                iu))
                      (narcout
                       (array-dimension aa
                        (1- (list-elt-index
                          pa (car ly)))))
                      (newshapeb
                       (append
                        (list narcout)
                        (array-dimensions
                                ab)))
                      (addy (array-transpose
                          (array-reshape
                           ab newshapeb)
                          ansy)))
                     (set-prob-array
                      nodeb
                      (list
                       (list-union
                        (list (car ly)) pb))
                      addy)
                     (set-descr
                      nodeb
                            (list
                       (list-union
                        (list (car ly)) db)))
                      (addarcs nodea nodeb)))
                )
              )
            )
          )
        )
      )
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; ID ACCESS
;
; Gets or sets nodal information in the id frame.

(defun get-prob-array (node)
  (car (fget current-id node 'data)))

(defun get-preds (node)
 (car (fget current-id node 'preds)))

(defun get-descr (node)
 (car (fget current-id node 'descr)))

(defun get-name (node)
 (fget current-id node 'name))

(defun set-prob-array (node preds array-address)
 (fset current-id node 'data array-address)
 (fset-list current-id node 'preds preds))

(defun set-preds (node preds)
  (fset-list current-id node 'preds preds))

(defun set-descr (node descr)
 (fset-list current-id node 'descr descr))

(defun print-node (node)
 (print "Node name is:")
 (print (get-name node))
 (print "Node predecessors are:")
 (print (get-preds node))
 (print "Node descriptors are:")
 (print (get-descr node))
 (print "Node probability array is:")
 (list (get-prob-array node)))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;  Test frames
;
(setq current-id 'id)
(fput 'id 'nodea 'type 'prob)
(fput 'id 'nodea 'preds '())
(fput 'id 'nodea 'descr '(a))
(fput 'id 'nodea 'name 'a)
(fput 'id 'nodea 'data (make-array '(2)
                :initial-contents
               '(0.7 0.3)))

(fput 'id 'nodeb 'type 'prob)
(fput 'id 'nodeb 'preds '(a))
(fput 'id 'nodeb 'descr '(a b))
(fput 'id 'nodeb 'name 'b)
(fput 'id 'nodeb 'data (make-array '(2 2)
                  :initial-contents
                '((0.6 0.4)
                 (0.5 0.5))))

(fput 'id1 'nodeg 'type 'prob)
(fput 'id1 'nodeg 'preds '(a b))
(fput 'id1 'nodeg 'descr '(a b g))
(fput 'id1 'nodeg 'name 'g)
(fput 'id1 'nodeg 'data (make-array '(2 2 2)
                  :initial-contents
                 '(((0.6 0.4)
                   (0.9 0.1))
                  ((0.2 0.8)
                  (0.7 0.3))) ))

(fput 'id1 'nodeh 'type 'prob)
(fput 'id1 'nodeh 'preds '(b c g))
(fput 'id1 'nodeh 'descr '(b c g h))
(fput 'id1 'nodeh 'name 'h)
(fput 'id1 'nodeh 'data (make-array '(2 2 2 2)
                  :initial-contents
                '((((0.6 0.4)
                    (0.5 0.5))
                   ((0.1 0.9)
                    (0.3 0.7)))
                  (;(0.9 0.1)
                    (0.2 0.8))
                   ((0.3 0.7)
                   (0.4 0.6)))) ))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;    FRAMES
;
; The following frames manipulation functions are from
;  Winston and Horn, "Lisp", second edition.
;

(defun fget (frame slot facet)
 (cdr (assoc facet (cdr (assoc slot (cdr (get frame
                                  'frame)))))))

(defun fput (frame slot facet value)
 (let ((value-list (follow-path (list slot facet)
                  (fget-frame frame))))
  (cond ((member value value-list) nil)
      (t (rplacd (last value-list) (list value))
        value))))

(defun fremove (frame slot facet value)
 (let ((value-list (follow-path (list slot facet)
                  (fget-frame frame))))
  (if (member value value-list)
    (delete value value-list))))


; fclear sets (<facet name> ....) to (<facet-name>).

(defun fclear (frame slot facet)
  (let ((clear-facet (follow-path (list slot facet)
                  (fget-frame frame))))
   (cond (clear-facet (rplacd clear-facet nil) t)
      (t nil))))

(defun fremove-slot (frame slot)
 (putprop frame
    (cons frame
       (remove (assoc slot (cdr (get frame 'frame)))
          (cdr (get frame 'frame))))
     'frame))


; FSET sets the addressed value slot to value rather than
;  adding value to the contents of value slot.  FSET-LIST
;  changes (facet <> <> <>) to
;  (eval (cons 'facet-name value-list)).
```

```
(defun fset (frame slot facet value) ; not W&H
 (let ((set-facet (follow-path (list slot facet)
                               (fget-frame frame))))
   (rplacd set-facet (list value))
    value))

(defun fset-list (frame slot facet value-list) ; not W&H
  (let ((set-facet (follow-path (list slot facet)
                                (fget-frame frame))))
   (rplacd set-facet value-list)
    value-list))

 (defun fget-frame (frame)
 (cond ((get frame 'frame))
       (t (setf (get frame 'frame) (list frame) ))))

(defun extend (key a-list)
 (cond ((assoc key (cdr a-list)))
       (t (cadr (rplacd (last a-list) (list (list
                                      key)))))))

(defun follow-path (path a-list)
 (cond ((null path) a-list)
       (t (follow-path (cdr path) (extend (car path)
                                      a-list)))))
```

LISP TOOLS

The following functions were used to implement the ID Solver's toolbox; however they form a valuable addition to any Lisp library.

List-transpose is a function on two lists, from-index and trans-index. Trans-index provides the receipe for moving the elements of from-index in a certain way. For example, (list-transpose '(a b c d) '(3 1 2 4)) will return (b c a d). The trans-list, '(3 1 2 4) can be understood in the following way:

1. Take 1st element of from-index and make it the new 3rd element.
2. Take 2nd element of from-index and make it the new 1st element.
3. Take 3rd element of from-index and make it the new 2nd element.
4. Take 4th element of from-index and make it the new 4th element.

```
(defun list-transpose (from-index trans-index)
 (list-transpose1 from-index trans-index
   (list-atoms (length from-index) 0)
           (iota (length from-index))))

(defun list-transpose1 (from-index trans-index
                        to-index index)
 (cond
 ((null (car index)) to-index)
 (t
  (setq to-index (list-index-assign
                  to-index
                  (car trans-index)
                   list-index-elt from-index
                     (car index))))
  (list-transpose1 from-index (cdr trans-index)
                   to-index (cdr index)))))
```

The functions char-less and char-greater are used to compare two character lists of equal length for alphabetical order

```
(defun char-less (c1 c2)
 (< (length (member c1 exploded-alphabet))
     (length (member c2 exploded-alphabet))))
(defun char-greater (c1 c2)
 (> (length (member c1 exploded-alphabet))
      (length (member c2 exploded-alphabet))))

(setq exploded-alphabet
 '(z y x w v u t s r q p o n m l k j i h g f e d c b a 9 8
   7 6 5 4 3 2 1 0))


; The function iota will generate a list of numbers, i.e.
; (iota 9) returns (1 2 3 4 5 6 7 8 9)

(defun iota (n) (iota1 n 1))

(defun iota1 (n cnt)
 (cond
 ((> cnt n) nil)
 (t (cons cnt (iota1 n (1+ cnt))))))

;;;;;;;;;;
;
; The macro mac-doarray will access an array, element by
;  element, and will perform the operations in the body.
;  The variable, indexlist, is the current index of the
;  array and is used in the body of commands. Indexlist
;  is iterated in row-major order. An example call is:
;     (mac-doarray a ((print indexlist)))
;  Notice that the body is a list of lists.
;
; The macro mac-doarray2 is similar to mac-doarray except
;  that it will access two arrays (not necessarily of the
;  same number of elements) by generating indexlist1 and
;  indexlist2 which correspond to arrayname1 and
;  arrayname2. Indexlist1 and indexlist2 are available
;  A typical call can be found in the definition of array-
;  sum-reduce.

(defmacro mac-doarray (arrayname body)
   `(let* ((targetlist (mapcar '1- (array-dimensions
                                   ,arrayname)))
         (i (length targetlist))
         (stoplist (list-index-assign
                  targetlist i
                  (+ (list-index-elt targetlist i)
                     1)))))
```

```
(do ((indexlist (list-atoms i 0)
                (check-indexlist
                 (list-index-assign
                      indexlist i
                      (+ (list-index-elt indexlist i)
                         1)) targetlist i)))
     ((list-equal indexlist stoplist) 't)
     ,@body)))

(defmacro mac-doarray2 (arrayname1 arrayname2 body)
`(let* ((targetlist1 (mapcar '1- (array-dimensions
                                   ,arrayname1)))
    (targetlist2 (mapcar '1- (array-dimensions
                               ,arrayname2)))
    (i1 (length targetlist1))
    (i2 (length targetlist2))
    (stoplist2 (list-index-assign
                targetlist2 i2
                 (+ (list-index-elt
                          targetlist2 i2) 1))))
(do ((indexlist1 (list-atoms i1 0)
                 (cond
                  ((list-equal indexlist1 targetlist1)
                               (list-atoms i1 0))
                 (t
                   (check-indexlist
                    (list-index-assign
                         indexlist1
                         i1
                         (+ (list-index-elt
                              indexlist1 i1) 1))
                    targetlist1 i1))))
    (indexlist2 (list-atoms i2 0)
                (check-indexlist
                 (list-index-assign
                      indexlist2
                      i2
                      (+ (list-index-elt
                          indexlist2 i2) 1))
                targetlist2 i2)))
     ((list-equal indexlist2 stoplist2) 't)
     ,@body)))

;;;;;;;;;;
;
; Check-indexlist is used in mac-doarray and mac-doarray2
;   to get the next valid indexlist in row-major-order.
```

```
(defun check-indexlist (indexlist targetlist i)
 (cond
 ((list-great-eqp indexlist targetlist) indexlist)
 ((not (list-less-eqp indexlist targetlist))
  (setq i (1- i))
   (setq indexlist (list-index-assign indexlist i
                              (+(list-index-elt
                                indexlist i) 1)))
   (setq indexlist (list-index-assign indexlist (+ i 1) 0))
   (check-indexlist indexlist targetlist i))
   (t
    indexlist)))

;;;;;;;;;;
;
; The function list-atoms will generate a list of the
;   specified atom (only one atom).  I.e. (list-atoms 3 'q)
;   returns (q q q).

(defun list-atoms (n a)
 (list-atoms1 n 1 a))

(defun list-atoms1 (n cnt a)
 (cond
  ((> cnt n) nil)
  (t (cons a (list-atoms1 n (1+ cnt) a)))))

;;;;;;;;;;
;
; Given a list and a valid number (index) list-index-elt
;   will return the corresponding element from the list.

 (defun list-index-elt (list index)
  (list-index-elt1 list index 1 ))

(defun list-index-elt1 (list index cnt )
 (cond
  ((= index cnt) (car list))
  (t (list-index-elt1 (cdr list) index (1+ cnt) ))))

;;;;;;;;;;
;
; Given a list and an element, list-elt-index
;   will return the corresponding index of the list.

(defun list-elt-index (list elt)
 (list-elt-index1 list elt 1 ))

(defun list-elt-index1 (list elt cnt )
 (cond
  ((equal elt (car list)) cnt)
  (t (list-elt-index1 (cdr list) elt (1+ cnt) ))))
```

```
;;;;;;;;;;
;
; List-index-assign will make the specified destructive
;   assignment to the indexed position of the list.

(defun list-index-assign (list index new)
 (do ((work-list list (cdr work-list))
      (work-index 1 (+ 1 work-index))
      (new-list nil))
     ((null work-list)  new-list)
     (cond
      ((= work-index index)
       (setq new-list (append new-list (list new))))
      (t (setq new-list (append new-list
                          (list (car work-list))))))))
;;;;;;;;;;
;
; Switch-last-two will return a list with the last two
;   elements of the given list reversed.

(defun switch-last-two (list1)
 (let* ((index1 (1- (length list1)))
        (index2 (length list1))
        (holder (list-index-elt list1 index1)))
  (list-index-assign
              (list-index-assign list1 index1
                  (list-index-elt list1 index2))
               index2
               holder)))

;;;;;;;;;;
;
; The next four functions are designed to make the
;   appropriate comparison, element by element, of two
;   lists of numbers.

(defun list-lessp (x y)
 (cond
  ((null (car x)) 't)
  ((< (car x) (car y))
   (list-lessp (cdr x) (cdr y)))))

(defun list-equal (x y)
 (cond
  ((not (= (length x) (length y))) nil)
  ((null (car x)) t)
  ((equal (car x) (car y))
   (list-equal (cdr x) (cdr y)))))
```

```
(defun list-less-eqp (x y)
  (cond
   ((null (car x)) 't)
   ((< (car x) (car y))
    (list-less-eqp (cdr x) (cdr y)))
   ((= (car x) (car y))
    (list-less-eqp (cdr x) (cdr y)))
   (t 'nil)))

(defun list-great-eqp (x y)
  (cond
   ((null (car x)) 't)
   ((> (car x) (car y))
    (list-great-eqp (cdr x) (cdr y)))
   ((= (car x) (car y))
    (list-great-eqp (cdr x) (cdr y)))
   (t 'nil)))

;;;;;;;;;;
;
; List-rotate will take the car of the given list and
;   append it to the cdr of the list, n times.

(defun list-rotate (n list1)
 (do ((x 1 (1+ x))
      (l list1 (cdr l))
     (res nil (append res (list (car l))) ))
     ((= x (1+ n)) (append l res))))

;;;;;;;;;;
;
; List-union will take two alphabetical lists and form
;   their union.  The result will be in alphabetical order.

(defun list-union (x y)
(cond ((equal x nil) y)
   ((equal y nil) x)
   (t  (our-sort (list-union1 x y) 'char-less))))

(defun list-union1 (x y)
  (cond ((null x)  y )
      ((member (car x) y) (list-union (cdr x) y))
      (t (cons (car x) (list-union (cdr x) y)))))

;;;;;;;;;;
;
; Our-sort and splice-in are defined in Winston and Horn,
;   2nd ed  p366
```

```
(defun our-sort (s predicate)
 (cond ((null s) nil)
     (t (splice-in (car s)
           (our-sort (cdr s) predicate)
           predicate))))

(defun splice-in (element s predicate)
 (cond ((null s) (list element))
     ((funcall predicate element (car s))
    (cons element s))
   (t (cons (car s) (splice-in element
                (cdr s)
                predicate)))))

        ;;;;;;;;;;;
;;;;;;;;;;;
;
; List-difference will return a result consisting of the
;   elements of "in" that are not members of the list,
;   "out".

(defun list-difference (in out)
 (cond ((null in) nil)
     ((member (car in) out) (list-difference (cdr in)
                                 out))
      (t (cons (car in) (list-difference (cdr in) out)))))

;;;;;;;;;;;
;;;;;;;;;;;
;
; A primitive on-line help file is found in Appendix B.
;


;;;;;;;;;;;
;;;;;;;;;;;
;
; Array-index-elt and array-index-assign are used
;   similarily as their counterpart commands for lists
;   (see above).

(defun array-index-elt (arrayname indexlist)
    (apply #'aref arrayname indexlist))

(defmacro array-index-assign (arrayname index new)
    `(setf (aref ,arrayname ,@index) ,new))

;;;;;;;;;;;
;;;;;;;;;;;
;
; Array-indexname-assign is the same as array-index-assign
;   except that the second argument is a variable that has
;   been set to a valid list.
```

```
(defun array-indexname-assign (arrayname indexname new)
   (eval (append (list 'setf (append '(aref)
                                (list arrayname)
                        indexname ))
                 (list 'new))))

;;;;;;;;;;
;;;;;;;;;;
;
; Given an array and an index, array-transpose will take
;  the array and rearrange the data according to the
;  index.  For a two-dimensional array, the only
;  transpose is the (2 1) transpose (switch rows and
;  columns).  A three-dimensional array can have six
;  different transpositions.  A typical call is:
;       (array-transpose a '(2 1 3))
;
; If a was a (2 3 3) shaped array, the (2 1 3) transpose
;  will return an array with shape, (3 2 3).
; This function is a version of the APL dyadic array
;  transpose.
(defun array-transpose (arrayname index)
 (let ((newarray (make-array (list-transpose
                (array-dimensions arrayname)
                   index)
          :initial-element '0)))
  (mac-doarray arrayname
   ((let ((translist (list-transpose indexlist index))
      (newelt (array-index-elt arrayname indexlist)))
     (array-indexname-assign newarray translist newelt))))
    newarray))

;;;;;;;;;;
;;;;;;;;;;
;
; Array-reshape will take the given array and the newshape
;  (list) and will make a new array with the newshape.
;  Elements from the given array will be stuffed into the
;  new array in row-major order.  Once the elements from
;  the given array are exhausted, the next elements will
;  come from the beginning of the array.  (The idea for
;  this function comes from APL.

(defun array-reshape (arrayname newshape)
 (let ((newarray (make-array newshape
                :initial-element '0)))
 (mac-doarray2 arrayname newarray
   ((array-indexname-assign newarray
                    indexlist2
                (array-index-elt arrayname
                     indexlist1))))
  newarray))
```

```
;;;;;;;;;;;
;
; Given two arrays of the same shape and size, array-mult-
;   elt will return an array which is the result of
;   multiplying the two arrays element by element.


(defun array-mult-elt (array1 array2)
  (let ((newarray (make-array (array-dimensions array1)
                  :initial-element '0)))
     (mac-doarray newarray
     ((let* ((elt1 (array-index-elt array1 indexlist))
          (elt2 (array-index-elt array2 indexlist))
       (newelt (* elt1 elt2)))
       (array-indexname-assign newarray indexlist
                               newelt))))
  newarray))

;;;;;;;;;;;
;
; Array-div-elt is the same as array-mult-elt only for
;   division.
 (defun array-div-elt (array1 array2)
  (let ((newarray (make-array (array-dimensions array1)
                  :initial-element '0)))
     (mac-doarray newarray
     ((let* ((elt1 (array-index-elt array1 indexlist))
          (elt2 (array-index-elt array2 indexlist))
       (newelt (/ elt1 elt2)))
       (array-indexname-assign newarray indexlist
                       newelt))))
  newarray))

;;;;;;;;;;
;
; List-index-delete will return a list with the indexed
;   element of the original list deleted.

(defun list-index-delete (list index)
  (list-index-delete1 (reverse list) index 1 (length list)))

(defun list-index-delete1 (list index cnt stop)
  (cond
    ((= cnt (+ stop 1)) (reverse list))
   ((= cnt index) (list-index-delete1 (cdr list)
                              index (1+ cnt) stop))
    (t (list-index-delete1 (append (cdr list)
          (list (car list)))
              index
              (1+ cnt)
              stop))))
```

```
;;;;;;;;;;
;;;;;;;;;;
;
; Array-sum-reduce will return an array that consists of
;   the sum of the given array across a specified direction
;   (of dimension).  In this case, dir is a number counted
;   from the right of the array dimension list.

(defun array-sum-reduce (arrayname dir)
 (let ((newarray (make-array
                  (list-index-delete
                     (array-dimensions arrayname)
                        dir)
                  :initial-element '0)))
  (mac-doarray2 newarray arrayname
   ((let* ((newindex (list-index-delete indexlist2 dir))
      (newelt1 (array-index-elt newarray newindex))
     (newelt2 (array-index-elt arrayname indexlist2)))
    (array-indexname-assign newarray newindex
                     (+ newelt1 newelt2)))))
 newarray))
```

APPENDEX B USER MANUAL

The software has been designed so that the analyst does not need to have a "User's Manual" to be able to use any or all commands. This appendix is a print of the help file, "helpt", in the influence diagram toolbox.

```
;;;;;;;;;;
;
; A primitive on-line help file.
;

(defun helpt ()

;     Top-Level Commands
(print '((arcrev nodea nodeb) - is the user level command
          which will reverse the probabilistic inference between
          nodes by adding arcs and performing Bayes rule))

;     Supporting Commands
(print '((addarcs nodea nodeb) - adds arcs to the
          predecessor (nodea) and the successor (nodeb) so that
          the descriptor list of nodea equals the predecessor list of nodeb
```

```
(print '((array-transpose array translist) - returns a new
        array consisting of the elements of array created by
        using translist as a mapping function to write to the
        new array.  In APL, this function is known as a dyadic
        transpose.))
(print '((array-indexname-assign arrayname indexname newelt)
          - makes the specified destructive assignment to the
          indexed position of the array.  Indexname is a variable
          that is a list which is a valid index to arrayname.))
(print '((array-index-assign arrayname index newelt) - makes
        the specified destructive assignment to the indexed
        position of the array.))
(print '((array-index-elt arrayname indexlist) - returns the
        element of the array (arrayname) located by the
        indexlist.))
(print '((array-reshape array newshape-list) - returns an
        array of new dimensions list, by using the elements of
        array1 in row major order.))
(print '((array-mult-elt array1 array2) - returns a new
        array formed by multiplying element by element array1
        and array2.))
(print '((array div-elt array1 array2) - returns a new array
        formed by dividing element by element array1 by
        array2.))
```

```
(print '((array-sum-reduce array right-dim) - returns an
        array by summing down the right-dim dimension of the
        array. Right-dim is an index of the dimension list
        denoted by counting from the right. The function is
        implemented in this way because the meaning of a given
        dimension is invariant to the length of the dimension
        list if counted from the right.))
(print '((char-less char1 char2) - compares two characters
        to see if char1 is alphabetically less than char2. ))
(print '((char-greater char1 char2) - compares two
        characters to see if char1 is alphabetically greater
        than char2.))
(print '((extend key a-list) - used in frames.))
(print '((fclear frame slot facet) - clears the values from
        the specified facet.))
(print '((fget frame slot facet) - returns the value
        associated with the named facet.))
(print '((fget-frame frame) - returns the frame.))
(print '((follow-path path a-list) - used in frames.))
(print '((fput frame slot facet value) - places the value in
        the frame.))
(print '((fremove frame slot facet value) - removes a value
        from the frame.
(print '((fremove-slot frame slot) - removes an entire
        slot.))
(print '((fset frame slot facet value) - destructively sets
```

```
        the value of the facet.))
(print '((fset-list frame slot facet value-list newelt)
        inserts newelt into the value-list (non-destructive).))
(print '((get-prob-array node) - gets the probability array
        at the node.))
(print '((get-preds node) - returns the predecessor list of
        the node.))
(print '((get-descr node) - returns the descriptor list of
        the node.))
(print '((get-name node) - gets the name of the node.))
(print '((iota n) - returns a list of the numbers 1 through n.
(print '((list-transpose list transform-list) - creates a
        list by applying the transform-list as a mapping
        function on the given list.))
(print '((list-atoms number atom) - returns a list of the
        length number consisting of the character atom.))
(print '((list-index-elt list index) - returns the element
        of the list at the indexed position.))
(print '((list-elt-index list elt) - returns an index
        corresponding to the position of the element in the
        list. This function will only return the first
        occurrence of the element.))
(print '((list-index-assign list index newatom) - a
        destructive assignment of newatom to the list by the
        index.))
```

```
(print '(list-index-delete list index) - erases the indexed
        element of the list. The list returned will be shorter
        by one element. ))
(print '(list-lessp list1 list2) - checks to see if the
        elements of list1 are less than the corresponding
        elements of list2.))
(print '(list-equal list1 list2) - checks to see if
        corresponding elements are equal.))
(print '(list-less-eqp list1 list2) - checks to see if the
        elements of list1 are less than or equal to the
        corresponding elements of list2.))
(print '(list-great-eqp list1 list2) - checks to see if the
        elements of list1 are greater than or equal to the
        corresponding elements of list2.))
(print '(list-union list1 list2) - returns an alphabetized
        list of atoms in either list1 or list2. ))
(print '(list-difference list1 list2) - returns list of
        atoms from list1 that are not in list2.))
(print '(list-rotate n list1) - appends the car of list to
        the cdr of list n number of times. ))
(print '(mac-doarray array (body)) - a construct that allow
        access of the array in row major order. Body is a list
        of commands that use the indexed element of the array.
        The variable that holds the current index is
        indexlist.))
```

```
(print '((mac-doarray2 array1 array2 (body)) - similiar to
        mac-doarray except that it will access two arrays (not
        necessarily of the same number of elements) by
        generating indexlist1 and indexlist2 which correspond
        to arrayname1 and arrayname2.  Indexlist1 and
        indexlist2 are available for use in the body of the
        call to mac-doarray2.))
(print '((print-node node) - displays the information of the
        node.))
(print '((reversex nodea nodeb) - will reverse the
        probabilistic inference between nodea and nodeb.  Nodea
        and Nodeb facets where the nodea and nodeb information
        is located))
(print '((set-prob-array node preds array-address) - assigns
        an array to a node in the influence diagram.
(print '((set-preds node preds) - assigns a predecessor list
        to a node in the influence diagram.))
 (print '((set-descr node descr) - assigns a descriptor list
        to a node in the influence diagram.))
(print '((switch-last-two list) - reverses the order of the
        two right-most elements of the list.))
't)
```

BIBLIOGRAPHY

Holtzman, Samuel. "Intelligent Decision Systems" PhD dissertation, 1985. Dept of Engineering - Economic Systems, Stanford University CA 94305.

Howard, Ronald A and James E. Matheson, eds. "Readings on The Principles and Applications of Decision Analysis." Vols I (1983) and II (1984). Strategic Decisions Group.

Olmstead, Scott M. "On Representing and Solving Decision Problems." PhD dissertation, 1984. Dept of Engineering - Economic Systems, Stanford University CA 94305.

Shachter, Ross D. "Evaluating Influence Diagrams" Submitted to Operations Research, Revised May, 1984. Dated January 1986. Department of Engineering - Economic Systems Stanford University, Stanford CA 94305.

Steele, Guy L. Jr. *Common Lisp: The Language*. 1984 Digital Press (Copyright by Digital Equipment Corporation).

Tatman, Joseph A. "Decision Processes in Influence Diagrams: Foundations and Analysis" PhD dissertation, 1985. Dept. of Engineering - Economic Systems, Stanford University CA 94305.

VITA

Captain Edward R. Dawson was born on 27 April 1957 at Elmendorf AFB, Alaska. He graduated from high school at London Central H.S. in High Wycombe, England, in 1975 and has attended the University of Southern Mississippi from which he received the degree of Bachelor of Science in Mathematics in August 1981. Upon graduation, he received a commission in the USAF through the ROTC program. He was subsequently assigned to Headquarters Tactical Air Command, Force Structure Analysis, Langley AFB, Virginia, until entering the School of Engineering, Air Force Institute of Technology, in May 1985.

Permanent address: Route 4, Box 866
South Boston, Virginia
24592

END

4-87

DTIC

END

4-87

DTIC